wolfSSL Documentation

2026-10-20

Contents

1 Introduction **21**
1.1 Why Choose wolfSSL? . . . 21

2 Building wolfSSL **22**
2.1 Getting wolfSSL Source Code . . . 22
2.2 Building on Unix-like Systems . . . 22
2.3 Building on Windows . . . 23
2.3.1 VS 2008 . . . 23
2.3.2 VS 2010 . . . 23
2.3.3 VS 2013 (64 bit solution) . . . 23
2.3.4 Cygwin . . . 23
2.4 Building with Various Vendor IDEs . . . 24
2.4.1 1. wolfSSL Library Build Project . . . 24
2.4.2 2. Application Executable Build Project . . . 25
2.5 Building in a non-standard environment . . . 25
2.5.1 Building into Yocto Linux . . . 25
2.5.2 Building with Atollic TrueSTUDIO . . . 26
2.5.3 Building with IAR . . . 26
2.5.4 Building on OS X and iOS . . . 27
2.5.5 Building with GCC ARM . . . 27
2.5.6 Building on Keil MDK-ARM . . . 28
2.6 Features Defined as C Pre-processor Macro . . . 28
2.6.1 Removing Features . . . 28
2.6.2 Enabling Features macros (on by default) . . . 39
2.6.3 Enabling Features Disabled by Default . . . 40
2.6.4 Customizing or Porting wolfSSL . . . 61
2.6.5 Reducing Memory or Code Usage . . . 67
2.6.6 Increasing Performance . . . 72
2.6.7 GCM Performance Tuning . . . 76
2.6.8 wolfSSL's Math Options . . . 76
2.6.9 Stack or Chip Specific Defines . . . 81
2.6.10 OS Specific Defines . . . 84
2.7 Build Options . . . 85
2.7.1 `--enable-debug` . . . 85
2.7.2 `--enable-distro` . . . 85
2.7.3 `--enable-singlethread` . . . 85
2.7.4 `--enable-dtls` . . . 85
2.7.5 `--disable-rng` . . . 85
2.7.6 `--enable-sctp` . . . 85
2.7.7 `--enable-openssh` . . . 85
2.7.8 `--enable-apachehttpd` . . . 85
2.7.9 `--enable-openvpn` . . . 86
2.7.10 `--enable-opensslextra` . . . 86
2.7.11 `--enable-opensslall` . . . 86
2.7.12 `--enable-maxstrength` . . . 86
2.7.13 `--disable-harden` . . . 86
2.7.14 `--enable-ipv6` . . . 86
2.7.15 `--enable-bump` . . . 86
2.7.16 `--enable-leanpsk` . . . 86
2.7.17 `--enable-leantls` . . . 87
2.7.18 `--enable-tinytls13` . . . 87

2.7.19 --enable-bigcache . . . 90
2.7.20 --enable-hugecache . . . 90
2.7.21 --enable-smallcache . . . 90
2.7.22 --enable-savesession . . . 90
2.7.23 --enable-savecert . . . 91
2.7.24 --enable-atomicuser . . . 91
2.7.25 --enable-pkcallbacks . . . 91
2.7.26 --enable-sniffer . . . 91
2.7.27 --enable-aesgcm . . . 91
2.7.28 --enable-aesccm . . . 91
2.7.29 --disable-aescbc . . . 91
2.7.30 --enable-aescfb . . . 92
2.7.31 --enable-aesctr . . . 92
2.7.32 --enable-aesni . . . 92
2.7.33 --enable-intelasm . . . 92
2.7.34 --enable-camellia . . . 92
2.7.35 --enable-md2 . . . 92
2.7.36 --enable-nullcipher . . . 92
2.7.37 --enable-ripemd . . . 92
2.7.38 --enable-blake2 . . . 92
2.7.39 --enable-blake2s . . . 92
2.7.40 --enable-sha3 . . . 93
2.7.41 --enable-sha512 . . . 93
2.7.42 --enable-she . . . 93
2.7.43 --enable-sessioncerts . . . 93
2.7.44 --enable-keygen . . . 93
2.7.45 --enable-certgen . . . 93
2.7.46 --enable-certext . . . 93
2.7.47 --enable-certreq . . . 93
2.7.48 --enable-sep . . . 94
2.7.49 --enable-hkdf . . . 94
2.7.50 --enable-x963kdf . . . 94
2.7.51 --enable-dsa . . . 94
2.7.52 --enable-eccshamir . . . 94
2.7.53 --enable-ecc . . . 94
2.7.54 --enable-ecccustcurves . . . 94
2.7.55 --enable-compkey . . . 94
2.7.56 --enable-curve25519 . . . 94
2.7.57 --enable-ed25519 . . . 95
2.7.58 --enable-fpecc . . . 95
2.7.59 --enable-eccencrypt . . . 95
2.7.60 --enable-psk . . . 95
2.7.61 --disable-errorstrings . . . 95
2.7.62 --disable-oldtls . . . 95
2.7.63 --enable-tlsv10 . . . 95
2.7.64 --enable-sslv3 . . . 95
2.7.65 --enable-stacksize . . . 95
2.7.66 --disable-memory . . . 95
2.7.67 --disable-rsa . . . 96
2.7.68 --enable-rsapss . . . 96
2.7.69 --disable-dh . . . 96
2.7.70 --enable-anon . . . 96
2.7.71 --disable-asn . . . 96
2.7.72 --disable-aes . . . 96

2.7.73 --disable-coding 96
2.7.74 --enable-base64encode 96
2.7.75 --disable-des3 96
2.7.76 --enable-arc4 96
2.7.77 --disable-md5 96
2.7.78 --disable-sha 96
2.7.79 --enable-webserver 97
2.7.80 --enable-fips 97
2.7.81 --enable-sha224 97
2.7.82 --disable-poly1305 97
2.7.83 --disable-chacha 97
2.7.84 --disable-hashdrbg 97
2.7.85 --disable-filesystem 97
2.7.86 --disable-inline 97
2.7.87 --enable-ocsp 97
2.7.88 --enable-ocspstapling 97
2.7.89 --enable-ocspstapling2 97
2.7.90 --enable-crl 98
2.7.91 --enable-crl-monitor 98
2.7.92 --enable-sni 98
2.7.93 --enable-maxfragment 98
2.7.94 --enable-alpn 98
2.7.95 --enable-truncatedhmac 98
2.7.96 --enable-renegotiation-indication 98
2.7.97 --enable-secure-renegotiation 98
2.7.98 --enable-supportedcurves 98
2.7.99 --enable-session-ticket 98
2.7.100--enable-extended-master 98
2.7.101--enable-tlsx 99
2.7.102--enable-pkcs7 99
2.7.103--enable-pkcs11 99
2.7.104--enable-ssh 99
2.7.105--enable-scep 99
2.7.106--enable-srp 99
2.7.107--enable-smallstack 99
2.7.108--enable-valgrind 99
2.7.109--enable-testcert 99
2.7.110--enable-iopool 99
2.7.111--enable-certservice 99
2.7.112--enable-jni 100
2.7.113--enable-lighty 100
2.7.114--enable-stunnel 100
2.7.115--enable-md4 100
2.7.116--enable-pwdbased 100
2.7.117--enable-scrypt 100
2.7.118--enable-cryptonly 100
2.7.119--disable-examples 100
2.7.120--disable-crypttests 100
2.7.121--enable-fast-rsa 100
2.7.122--enable-staticmemory 101
2.7.123--enable-mcapi 101
2.7.124--enable-asynccrypt 101
2.7.125--enable-sessionexport 101
2.7.126--enable-aeskeywrap 101

2.7.127 --enable-jobserver . . . 101
2.7.128 --enable-shared[=PKGS] . . . 101
2.7.129 --enable-static[=PKGS] . . . 101
2.7.130 --with-liboqs=PATH . . . 101
2.7.131 --with-libz=PATH . . . 101
2.7.132 --with-cavium . . . 102
2.7.133 --with-user-crypto . . . 102
2.7.134 --enable-rsavfy . . . 102
2.7.135 --enable-rsapub . . . 102
2.7.136 --enable-armasm . . . 102
2.7.137 --disable-tlsv12 . . . 102
2.7.138 --enable-tls13 . . . 102
2.7.139 --enable-all . . . 102
2.7.140 --enable-xts . . . 102
2.7.141 --enable-asio . . . 102
2.7.142 --enable-qt . . . 103
2.7.143 --enable-qt-test . . . 103
2.7.144 --enable-apache-httpd . . . 103
2.7.145 --enable-afalg . . . 103
2.7.146 --enable-devcrypto . . . 103
2.7.147 --enable-mcast . . . 103
2.7.148 --disable-pkcs12 . . . 103
2.7.149 --enable-fallback-scsv . . . 103
2.7.150 --enable-psk-one-id . . . 103
2.7.151 --enable-cryptocb . . . 104
2.7.152 --enable-reproducible-build . . . 104
2.7.153 --enable-sys-ca-certs . . . 104
2.8 Special Math Optimization Flags . . . 104
2.8.1 --enable-fastmath . . . 104
2.8.2 --enable-fasthugemath . . . 104
2.8.3 --enable-sp-math . . . 104
2.8.4 --enable-sp-math-all . . . 105
2.8.5 --enable-sp-asm . . . 105
2.8.6 --enable-sp=OPT . . . 105
2.9 Cross Compiling . . . 109
2.9.1 Example cross compile configure options for toolchain builds . . . 109
2.10 Building Ports . . . 111
2.11 Building For NXP CAAM . . . 112
2.11.1 i.MX8 (Linux) . . . 112
2.11.2 i.MX8 (QNX) . . . 122
2.11.3 i.MX6 (QNX) . . . 122
2.11.4 IMXRT1170 (FreeRTOS) . . . 122

3 Getting Started 124
3.1 General Description . . . 124
3.2 Testsuite . . . 124
3.3 Client Example . . . 126
3.4 Server Example . . . 128
3.5 EchoServer Example . . . 129
3.6 EchoClient Example . . . 130
3.7 Benchmark . . . 130
3.7.1 Relative Performance . . . 132
3.7.2 Benchmarking Notes . . . 132
3.7.3 Benchmarking on Embedded Systems . . . 134

3.8 Changing a Client Application to Use wolfSSL . . . 135
3.9 Changing a Server Application to Use wolfSSL . . . 136

4 Features 138
4.1 Features Overview . . . 138
4.2 Protocol Support . . . 138
4.2.1 Server Functions . . . 138
4.2.2 Client Functions . . . 138
4.2.3 Robust Client and Server Downgrade . . . 139
4.2.4 IPv6 Support . . . 139
4.2.5 DTLS . . . 139
4.2.6 LwIP (Lightweight Internet Protocol) . . . 139
4.2.7 TLS Extensions . . . 140
4.3 Cipher Support . . . 140
4.3.1 Cipher Suite Strength and Choosing Proper Key Sizes . . . 140
4.3.2 Supported Cipher Suites . . . 141
4.3.3 AEAD Suites . . . 144
4.3.4 Block and Stream Ciphers . . . 144
4.3.5 Hashing Functions . . . 145
4.3.6 Public Key Options . . . 145
4.3.7 ECC Support . . . 145
4.3.8 PKCS Support . . . 146
4.3.9 Forcing the Use of a Specific Cipher . . . 150
4.3.10 OpenQuantumSafe's liboqs Integration . . . 150
4.4 Hardware Accelerated Crypto . . . 150
4.4.1 AES-NI . . . 150
4.4.2 STM32F2 . . . 151
4.4.3 Cavium NITROX . . . 151
4.4.4 ESP32-WROOM-32 . . . 151
4.4.5 ESP8266 . . . 151
4.4.6 EFR32 . . . 151
4.4.7 MAX32665/MAX32666 . . . 152
4.5 SSL Inspection (Sniffer) . . . 152
4.6 Static Buffer Allocation Option . . . 153
4.6.1 Basic Operation of Static Buffer Allocation . . . 153
4.6.2 Specifying Static Buffer Use . . . 154
4.6.3 Enabling Static Buffer Allocation . . . 154
4.6.4 Using Static Buffer Allocation . . . 156
4.6.5 Using Static Memory Without TLS (Crypto-Only) . . . 157
4.6.6 Adjustment of Static Buffer Allocation . . . 159
4.7 Compression . . . 162
4.8 Pre-Shared Keys . . . 162
4.9 Client Authentication . . . 163
4.10 Server Name Indication . . . 163
4.11 Handshake Modifications . . . 164
4.11.1 Grouping Handshake Messages . . . 164
4.12 Truncated HMAC . . . 164
4.13 Timing-Resistance in wolfSSL . . . 164
4.14 Fixed ABI . . . 165

5 Portability 167
5.1 Abstraction Layers . . . 167
5.1.1 C Standard Library Abstraction Layer . . . 167
5.1.2 Custom Input/Output Abstraction Layer . . . 168

5.1.3 Operating System Abstraction Layer . . . 168
5.2 Supported Operating Systems . . . 168
5.3 Supported Chipmakers . . . 169
5.4 C# Wrapper . . . 169

6 Callbacks 171
6.1 HandShake Callback . . . 171
6.2 Timeout Callback . . . 171
6.3 User Atomic Record Layer Processing . . . 172
6.4 Public Key Callbacks . . . 173
6.4.1 DH Callbacks . . . 173
6.4.2 Ed25519 Callbacks . . . 173
6.4.3 X25519 Callbacks . . . 174
6.4.4 Ed448 Callbacks . . . 175
6.4.5 X448 Callbacks . . . 175
6.4.6 RSA PSS Callbacks . . . 176
6.4.7 ECC Callbacks . . . 177
6.5 Crypto Callbacks (cryptocb) . . . 178
6.5.1 Using Crypto callbacks . . . 178
6.5.2 Writing your crypto callback . . . 179
6.5.3 Handling the request . . . 181
6.5.4 Troubleshooting . . . 181
6.5.5 Examples . . . 182

7 Keys and Certificates 183
7.1 Supported Formats and Sizes . . . 183
7.2 Supported Certificate Extensions . . . 183
7.3 Certificate Loading . . . 185
7.3.1 Loading CA Certificates** . . . 185
7.3.2 Loading Client or Server Certificates . . . 185
7.3.3 Loading Private Keys . . . 186
7.3.4 Loading Trusted Peer Certificates . . . 186
7.4 Certificate Chain Verification . . . 186
7.5 Domain Name Check for Server Certificates . . . 187
7.6 No File System and using Certificates . . . 187
7.6.1 Test Certificate and Key Buffers . . . 187
7.7 Serial Number Retrieval . . . 187
7.8 RSA Key Generation . . . 188
7.8.1 RSA Key Generation Notes . . . 189
7.9 Certificate Generation . . . 189
7.10 Certificate Signing Request (CSR) Generation . . . 191
7.10.1 Limitations . . . 193
7.11 Convert to raw ECC key . . . 193
7.11.1 Example . . . 193

8 Debugging 194
8.1 Debugging and Logging . . . 194
8.2 Error Codes . . . 194

9 Library Design 195
9.1 Library Headers . . . 195
9.2 Startup and Exit . . . 195
9.3 Structure Usage . . . 195
9.4 Thread Safety . . . 195
9.5 Input and Output Buffers . . . 196

COPYRIGHT ©2024 wolfSSL Inc.

10 wolfCrypt Usage Reference **197**
10.1 Hash Functions . . . 197
10.1.1 MD4 . . . 197
10.1.2 MD5 . . . 197
10.1.3 SHA / SHA-224 / SHA-256 / SHA-384 / SHA-512 . . . 198
10.1.4 BLAKE2b . . . 198
10.1.5 RIPEMD-160 . . . 199
10.2 Keyed Hash Functions . . . 199
10.2.1 HMAC . . . 199
10.2.2 GMAC . . . 199
10.2.3 Poly1305 . . . 200
10.3 Block Ciphers . . . 200
10.3.1 AES . . . 200
10.4 Stream Ciphers . . . 202
10.4.1 ARC4 . . . 202
10.4.2 ChaCha . . . 202
10.5 Public Key Cryptography . . . 203
10.5.1 RSA . . . 203
10.5.2 DH (Diffie-Hellman) . . . 204
10.5.3 EDH (Ephemeral Diffie-Hellman) . . . 205
10.5.4 DSA (Digital Signature Algorithm) . . . 205

11 SSL Tutorial **207**
11.1 Introduction . . . 207
11.1.1 Examples Used in this Tutorial . . . 207
11.2 Quick Summary of SSL/TLS . . . 207
11.3 Getting the Source Code . . . 207
11.4 Base Example Modifications . . . 208
11.4.1 Modifications to the echoserver (tcpserv04.c) . . . 208
11.4.2 Modifications to the echoclient (tcpcli01.c) . . . 208
11.4.3 Modifications to unp.h header . . . 208
11.5 Building and Installing wolfSSL . . . 209
11.6 Initial Compilation . . . 211
11.7 Libraries . . . 211
11.8 Headers . . . 211
11.9 Startup/Shutdown . . . 212
11.10 WOLFSSL_CTX Factory . . . 212
11.11 WOLFSSL Object . . . 214
11.11.1 EchoClient . . . 214
11.11.2 EchoServer . . . 214
11.12 Sending/Receiving Data . . . 215
11.12.1 Sending with EchoClient . . . 215
11.12.2 Receiving with EchoServer . . . 216
11.13 Signal Handling . . . 216
11.13.1 Echoclient / Echoserver . . . 216
11.14 Certificates . . . 218
11.15 Conclusion . . . 218

12 Best Practices for Embedded Devices **219**
12.1 Creating Private Keys . . . 219
12.2 Digitally Signing and Authenticating with wolfSSL . . . 219

13 OpenSSL Compatibility **220**
13.1 Compatibility with OpenSSL . . . 220
13.2 Differences Between wolfSSL and OpenSSL . . . 220

COPYRIGHT ©2024 wolfSSL Inc.

13.3 Supported OpenSSL Structures . . . 221
13.4 Supported OpenSSL Functions . . . 221
13.5 x509 Certificates . . . 222

14 Licensing 223
14.1 Open Source . . . 223
14.2 Commercial Licensing . . . 223
14.3 FIPS 140-2/3 Validation . . . 223
14.4 Support Packages . . . 223

15 Support and Consulting 224
15.1 How to Get Support . . . 224
15.1.1 Bugs Reports and Support Issues . . . 224
15.2 Consulting . . . 224
15.2.1 Feature Additions and Porting . . . 224
15.2.2 Competitive Upgrade Program . . . 224
15.2.3 Design Consulting . . . 225

16 wolfSSL (formerly CyaSSL) Updates 226
16.1 Product Release Information . . . 226

A wolfSSL API Reference 227
A.1 CertManager API . . . 227
A.1.1 Functions . . . 227
A.1.2 Functions Documentation . . . 229
A.2 Memory Handling . . . 248
A.2.1 Functions . . . 248
A.2.2 Functions Documentation . . . 254
A.3 OpenSSL API . . . 272
A.3.1 Functions . . . 272
A.3.2 Functions Documentation . . . 277
A.4 wolfSSL Certificates and Keys . . . 300
A.4.1 Functions . . . 300
A.4.2 Functions Documentation . . . 313
A.5 wolfSSL Connection, Session, and I/O . . . 383
A.5.1 Functions . . . 384
A.5.2 Functions Documentation . . . 400
A.6 wolfSSL Context and Session Set Up . . . 479
A.6.1 Types . . . 479
A.6.2 Functions . . . 479
A.6.3 Types Documentation . . . 509
A.6.4 Functions Documentation . . . 510
A.7 wolfSSL Error Handling and Reporting . . . 625
A.7.1 Functions . . . 625
A.7.2 Functions Documentation . . . 627
A.8 wolfSSL Initialization/Shutdown . . . 635
A.8.1 Functions . . . 635
A.8.2 Functions Documentation . . . 639

B wolfCrypt API Reference 662
B.1 ASN.1 . . . 662
B.1.1 Functions . . . 662
B.1.2 Functions Documentation . . . 673
B.2 Base Encoding . . . 756
B.2.1 Functions . . . 756

COPYRIGHT ©2024 wolfSSL Inc.

B.2.2 Functions Documentation . . . 758
B.3 Compression . . . 764
B.3.1 Functions . . . 764
B.3.2 Functions Documentation . . . 765
B.4 Error Reporting . . . 770
B.4.1 Functions . . . 770
B.4.2 Functions Documentation . . . 770
B.5 IoT-Safe Module . . . 771
B.5.1 Functions . . . 771
B.5.2 Detailed Description . . . 774
B.5.3 Functions Documentation . . . 774
B.6 Logging . . . 787
B.6.1 Functions . . . 787
B.6.2 Functions Documentation . . . 788
B.7 Math API . . . 789
B.7.1 Functions . . . 789
B.7.2 Functions Documentation . . . 789
B.8 Random Number Generation . . . 790
B.8.1 Functions . . . 790
B.8.2 Detailed Description . . . 804
B.8.3 Functions Documentation . . . 804
B.9 Signature API . . . 869
B.9.1 Functions . . . 869
B.9.2 Functions Documentation . . . 871
B.10 wolfCrypt Init and Cleanup . . . 879
B.10.1 Functions . . . 879
B.10.2 Functions Documentation . . . 880
B.11 Algorithms - 3DES . . . 889
B.11.1 Functions . . . 889
B.11.2 Functions Documentation . . . 892
B.12 Algorithms - AES . . . 906
B.12.1 Functions . . . 906
B.12.2 Functions Documentation . . . 921
B.13 Algorithms - ARC4 . . . 1007
B.13.1 Functions . . . 1007
B.13.2 Functions Documentation . . . 1008
B.14 Algorithms - BLAKE2 . . . 1011
B.14.1 Functions . . . 1011
B.14.2 Functions Documentation . . . 1012
B.15 Algorithms - Camellia . . . 1021
B.15.1 Functions . . . 1021
B.15.2 Functions Documentation . . . 1022
B.16 Algorithms - ChaCha . . . 1026
B.16.1 Functions . . . 1026
B.16.2 Functions Documentation . . . 1026
B.17 Algorithms - ChaCha20_Poly1305 . . . 1030
B.17.1 Functions . . . 1030
B.17.2 Functions Documentation . . . 1032
B.18 Algorithm - CMAC . . . 1043
B.18.1 Functions . . . 1043
B.18.2 Functions Documentation . . . 1045
B.19 Callbacks - CryptoCb . . . 1053
B.19.1 Functions . . . 1053
B.19.2 Functions Documentation . . . 1054

B.20 Algorithms - Curve25519 . . . 1056
B.20.1 Functions . . . 1056
B.20.2 Functions Documentation . . . 1059
B.21 Algorithms - Curve448 . . . 1083
B.21.1 Functions . . . 1083
B.21.2 Functions Documentation . . . 1086
B.22 Algorithms - DSA . . . 1109
B.22.1 Functions . . . 1109
B.22.2 Functions Documentation . . . 1111
B.23 Algorithms - Diffie-Hellman . . . 1128
B.23.1 Functions . . . 1128
B.23.2 Functions Documentation . . . 1131
B.24 Algorithms - ECC . . . 1156
B.24.1 Functions . . . 1156
B.24.2 Functions Documentation . . . 1166
B.25 Algorithms - ED25519 . . . 1244
B.25.1 Functions . . . 1244
B.25.2 Functions Documentation . . . 1248
B.26 Algorithms - ED448 . . . 1278
B.26.1 Functions . . . 1278
B.26.2 Functions Documentation . . . 1281
B.27 Overview of ECCSI . . . 1303
B.27.1 Detailed Description . . . 1303
B.28 Setup ECCSI Key . . . 1304
B.28.1 Functions . . . 1304
B.28.2 Detailed Description . . . 1305
B.28.3 Functions Documentation . . . 1305
B.29 Operations for Signing and Verifying with ECCSI Key . . . 1307
B.29.1 Functions . . . 1307
B.29.2 Detailed Description . . . 1308
B.29.3 Functions Documentation . . . 1308
B.30 Overview of SAKKE Key . . . 1311
B.30.1 Detailed Description . . . 1311
B.31 Setup SAKKE Key . . . 1312
B.31.1 Functions . . . 1312
B.31.2 Detailed Description . . . 1313
B.31.3 Functions Documentation . . . 1313
B.32 Operations on/with SAKKE RSK . . . 1315
B.32.1 Functions . . . 1315
B.32.2 Detailed Description . . . 1316
B.32.3 Functions Documentation . . . 1316
B.33 Operations using SAKKE Key . . . 1317
B.33.1 Functions . . . 1317
B.33.2 Detailed Description . . . 1318
B.33.3 Functions Documentation . . . 1318
B.34 Algorithms - HMAC . . . 1320
B.34.1 Functions . . . 1320
B.34.2 Functions Documentation . . . 1322
B.35 Algorithms - LMS / HSS (RFC 8554) . . . 1336
B.35.1 Functions . . . 1336
B.35.2 Detailed Description . . . 1339
B.35.3 Functions Documentation . . . 1339
B.36 Algorithms - MD2 . . . 1355
B.36.1 Functions . . . 1355

B.36.2 Functions Documentation . . . 1355
B.37 Algorithms - MD4 . . . 1358
B.37.1 Functions . . . 1358
B.37.2 Functions Documentation . . . 1358
B.38 Algorithms - MD5 . . . 1361
B.38.1 Functions . . . 1361
B.38.2 Functions Documentation . . . 1361
B.39 Algorithms - ML-DSA (FIPS 204) . . . 1366
B.39.1 Functions . . . 1366
B.39.2 Detailed Description . . . 1372
B.39.3 Functions Documentation . . . 1372
B.40 Algorithms - ML-KEM (FIPS 203) . . . 1397
B.40.1 Functions . . . 1397
B.40.2 Detailed Description . . . 1400
B.40.3 Functions Documentation . . . 1400
B.41 Algorithms - Password Based . . . 1414
B.41.1 Functions . . . 1414
B.41.2 Functions Documentation . . . 1416
B.42 Algorithms - PKCS7 . . . 1425
B.42.1 Types . . . 1425
B.42.2 Functions . . . 1425
B.42.3 Types Documentation . . . 1431
B.42.4 Functions Documentation . . . 1432
B.43 Algorithms - PKCS11 . . . 1480
B.43.1 Functions . . . 1480
B.43.2 Functions Documentation . . . 1481
B.44 Algorithms - PKCS12 . . . 1482
B.44.1 Functions . . . 1482
B.44.2 Detailed Description . . . 1485
B.44.3 Functions Documentation . . . 1485
B.45 Algorithms - Poly1305 . . . 1495
B.45.1 Functions . . . 1496
B.45.2 Functions Documentation . . . 1496
B.46 Platform Security Architecture (PSA) API . . . 1501
B.46.1 Functions . . . 1501
B.46.2 Functions Documentation . . . 1502
B.47 Algorithms - PUF . . . 1505
B.47.1 Functions . . . 1506
B.47.2 Functions Documentation . . . 1507
B.48 Algorithms - RIPEMD . . . 1517
B.48.1 Functions . . . 1517
B.48.2 Functions Documentation . . . 1517
B.49 Algorithms - RSA . . . 1519
B.49.1 Functions . . . 1519
B.49.2 Functions Documentation . . . 1526
B.50 Algorithms - SHA 128/224/256/384/512 . . . 1579
B.50.1 Functions . . . 1579
B.50.2 Functions Documentation . . . 1592
B.51 Algorithms - SHE . . . 1701
B.51.1 Functions . . . 1701
B.51.2 Functions Documentation . . . 1705
B.52 Algorithm - SipHash . . . 1724
B.52.1 Functions . . . 1724
B.52.2 Functions Documentation . . . 1724

B.53 Algorithms - SLH-DSA (FIPS 205) . . . 1727
B.53.1 Functions . . . 1728
B.53.2 Detailed Description . . . 1732
B.53.3 Functions Documentation . . . 1732
B.54 Algorithm - SRTP KDF . . . 1758
B.54.1 Functions . . . 1758
B.54.2 Functions Documentation . . . 1758
B.55 Algorithms - SRP . . . 1764
B.55.1 Functions . . . 1764
B.55.2 Functions Documentation . . . 1765
B.56 Algorithms - XMSS / XMSS^MT (RFC 8391) . . . 1777
B.56.1 Functions . . . 1777
B.56.2 Detailed Description . . . 1780
B.56.3 Functions Documentation . . . 1780

C API Header Files 1794
C.1 aes.h . . . 1794
C.1.1 Functions . . . 1794
C.1.2 Functions Documentation . . . 1807
C.1.3 Source code . . . 1882
C.2 arc4.h . . . 1888
C.2.1 Functions . . . 1888
C.2.2 Functions Documentation . . . 1888
C.2.3 Source code . . . 1891
C.3 ascon.h . . . 1891
C.3.1 Functions . . . 1891
C.3.2 Functions Documentation . . . 1892
C.3.3 Source code . . . 1903
C.4 asn.h . . . 1904
C.4.1 Functions . . . 1904
C.4.2 Functions Documentation . . . 1905
C.4.3 Source code . . . 1911
C.5 asn_public.h . . . 1912
C.5.1 Functions . . . 1912
C.5.2 Functions Documentation . . . 1926
C.5.3 Source code . . . 2011
C.6 blake2.h . . . 2017
C.6.1 Functions . . . 2017
C.6.2 Functions Documentation . . . 2019
C.6.3 Source code . . . 2027
C.7 bn.h . . . 2028
C.7.1 Functions . . . 2028
C.7.2 Functions Documentation . . . 2028
C.7.3 Source code . . . 2029
C.8 camellia.h . . . 2029
C.8.1 Functions . . . 2029
C.8.2 Functions Documentation . . . 2029
C.8.3 Source code . . . 2033
C.9 chacha20_poly1305.h . . . 2033
C.9.1 Functions . . . 2033
C.9.2 Functions Documentation . . . 2036
C.9.3 Source code . . . 2046
C.10 chacha.h . . . 2047
C.10.1 Functions . . . 2047

COPYRIGHT ©2024 wolfSSL Inc.

C.10.2 Functions Documentation . . . 2048
C.10.3 Source code . . . 2051
C.11 cmac.h . . . 2051
C.11.1 Functions . . . 2051
C.11.2 Functions Documentation . . . 2052
C.11.3 Source code . . . 2059
C.12 coding.h . . . 2060
C.12.1 Functions . . . 2060
C.12.2 Functions Documentation . . . 2062
C.12.3 Source code . . . 2068
C.13 compress.h . . . 2068
C.13.1 Functions . . . 2068
C.13.2 Functions Documentation . . . 2069
C.13.3 Source code . . . 2073
C.14 cryptocb.h . . . 2073
C.14.1 Functions . . . 2073
C.14.2 Functions Documentation . . . 2075
C.14.3 Source code . . . 2083
C.15 curve25519.h . . . 2084
C.15.1 Functions . . . 2084
C.15.2 Functions Documentation . . . 2087
C.15.3 Source code . . . 2108
C.16 curve448.h . . . 2109
C.16.1 Functions . . . 2109
C.16.2 Functions Documentation . . . 2112
C.16.3 Source code . . . 2130
C.17 des3.h . . . 2131
C.17.1 Functions . . . 2131
C.17.2 Functions Documentation . . . 2133
C.17.3 Source code . . . 2143
C.18 dh.h . . . 2143
C.18.1 Functions . . . 2143
C.18.2 Functions Documentation . . . 2147
C.18.3 Source code . . . 2168
C.19 doxygen_groups.h . . . 2171
C.20 doxygen_pages.h . . . 2171
C.21 dsa.h . . . 2171
C.21.1 Functions . . . 2171
C.21.2 Functions Documentation . . . 2173
C.21.3 Source code . . . 2185
C.22 ecc.h . . . 2186
C.22.1 Functions . . . 2186
C.22.2 Functions Documentation . . . 2196
C.22.3 Source code . . . 2260
C.23 eccsi.h . . . 2265
C.23.1 Functions . . . 2265
C.23.2 Functions Documentation . . . 2267
C.23.3 Source code . . . 2271
C.24 ed25519.h . . . 2272
C.24.1 Functions . . . 2272
C.24.2 Functions Documentation . . . 2277
C.24.3 Source code . . . 2302
C.25 ed448.h . . . 2304
C.25.1 Functions . . . 2304

C.25.2 Functions Documentation . . . 2308
C.25.3 Source code . . . 2327
C.26 error-crypt.h . . . 2328
C.26.1 Functions . . . 2328
C.26.2 Functions Documentation . . . 2328
C.26.3 Source code . . . 2329
C.27 evp.h . . . 2329
C.27.1 Functions . . . 2329
C.27.2 Functions Documentation . . . 2331
C.27.3 Source code . . . 2340
C.28 hash.h . . . 2341
C.28.1 Functions . . . 2341
C.28.2 Functions Documentation . . . 2342
C.28.3 Source code . . . 2354
C.29 hmac.h . . . 2355
C.29.1 Functions . . . 2355
C.29.2 Functions Documentation . . . 2357
C.29.3 Source code . . . 2369
C.30 iotsafe.h . . . 2371
C.30.1 Functions . . . 2371
C.30.2 Functions Documentation . . . 2374
C.30.3 Source code . . . 2385
C.31 kdf.h . . . 2386
C.31.1 Functions . . . 2386
C.31.2 Functions Documentation . . . 2387
C.31.3 Source code . . . 2392
C.32 logging.h . . . 2393
C.32.1 Functions . . . 2393
C.32.2 Functions Documentation . . . 2393
C.32.3 Source code . . . 2395
C.33 md2.h . . . 2395
C.33.1 Functions . . . 2395
C.33.2 Functions Documentation . . . 2395
C.33.3 Source code . . . 2398
C.34 md4.h . . . 2398
C.34.1 Functions . . . 2398
C.34.2 Functions Documentation . . . 2398
C.34.3 Source code . . . 2400
C.35 md5.h . . . 2400
C.35.1 Functions . . . 2400
C.35.2 Functions Documentation . . . 2401
C.35.3 Source code . . . 2404
C.36 memory.h . . . 2404
C.36.1 Functions . . . 2404
C.36.2 Functions Documentation . . . 2407
C.36.3 Source code . . . 2420
C.37 ocsp.h . . . 2420
C.37.1 Functions . . . 2420
C.37.2 Functions Documentation . . . 2421
C.37.3 Source code . . . 2429
C.38 pem.h . . . 2430
C.38.1 Functions . . . 2430
C.38.2 Functions Documentation . . . 2430
C.38.3 Source code . . . 2431

C.39 pkcs11.h . . . 2431
C.39.1 Functions . . . 2431
C.39.2 Functions Documentation . . . 2431
C.39.3 Source code . . . 2432
C.40 pkcs12.h . . . 2433
C.40.1 Functions . . . 2433
C.40.2 Functions Documentation . . . 2436
C.40.3 Source code . . . 2445
C.41 pkcs7.h . . . 2446
C.41.1 Types . . . 2446
C.41.2 Functions . . . 2446
C.41.3 Types Documentation . . . 2452
C.41.4 Functions Documentation . . . 2453
C.41.5 Source code . . . 2495
C.42 poly1305.h . . . 2499
C.42.1 Functions . . . 2499
C.42.2 Functions Documentation . . . 2500
C.42.3 Source code . . . 2504
C.43 psa.h . . . 2504
C.43.1 Functions . . . 2504
C.43.2 Functions Documentation . . . 2505
C.43.3 Source code . . . 2508
C.44 puf.h . . . 2509
C.44.1 Functions . . . 2509
C.44.2 Functions Documentation . . . 2510
C.44.3 Source code . . . 2518
C.45 pwdbased.h . . . 2519
C.45.1 Functions . . . 2519
C.45.2 Functions Documentation . . . 2520
C.45.3 Source code . . . 2528
C.46 quic.h . . . 2529
C.46.1 Functions . . . 2529
C.46.2 Attributes . . . 2531
C.46.3 Functions Documentation . . . 2532
C.46.4 Attributes Documentation . . . 2544
C.46.5 Source code . . . 2545
C.47 random.h . . . 2547
C.47.1 Functions . . . 2547
C.47.2 Functions Documentation . . . 2556
C.47.3 Source code . . . 2594
C.48 ripemd.h . . . 2597
C.48.1 Functions . . . 2597
C.48.2 Functions Documentation . . . 2598
C.48.3 Source code . . . 2599
C.49 rsa.h . . . 2600
C.49.1 Functions . . . 2600
C.49.2 Functions Documentation . . . 2606
C.49.3 Source code . . . 2650
C.50 sakke.h . . . 2653
C.50.1 Functions . . . 2653
C.50.2 Functions Documentation . . . 2655
C.50.3 Source code . . . 2659
C.51 sha256.h . . . 2661
C.51.1 Functions . . . 2661

C.51.2 Functions Documentation . . . 2663
C.51.3 Source code . . . 2676
C.52 sha3.h . . . 2677
C.52.1 Functions . . . 2677
C.52.2 Functions Documentation . . . 2684
C.52.3 Source code . . . 2729
C.53 sha512.h . . . 2733
C.53.1 Functions . . . 2733
C.53.2 Functions Documentation . . . 2736
C.53.3 Source code . . . 2759
C.54 sha.h . . . 2762
C.54.1 Functions . . . 2762
C.54.2 Functions Documentation . . . 2763
C.54.3 Source code . . . 2768
C.55 signature.h . . . 2768
C.55.1 Functions . . . 2768
C.55.2 Functions Documentation . . . 2770
C.55.3 Source code . . . 2777
C.56 siphash.h . . . 2778
C.56.1 Functions . . . 2778
C.56.2 Functions Documentation . . . 2778
C.56.3 Source code . . . 2781
C.57 srp.h . . . 2782
C.57.1 Functions . . . 2782
C.57.2 Functions Documentation . . . 2783
C.57.3 Source code . . . 2793
C.58 ssl.h . . . 2794
C.58.1 Types . . . 2794
C.58.2 Functions . . . 2794
C.58.3 Types Documentation . . . 2883
C.58.4 Functions Documentation . . . 2883
C.58.5 Source code . . . 3223
C.59 tfm.h . . . 3247
C.59.1 Functions . . . 3247
C.59.2 Functions Documentation . . . 3247
C.59.3 Source code . . . 3248
C.60 types.h . . . 3248
C.60.1 Functions . . . 3248
C.60.2 Functions Documentation . . . 3251
C.60.3 Source code . . . 3261
C.61 wc_encrypt.h . . . 3262
C.61.1 Functions . . . 3262
C.61.2 Functions Documentation . . . 3264
C.61.3 Source code . . . 3271
C.62 wc_lms.h . . . 3272
C.62.1 Functions . . . 3272
C.62.2 Functions Documentation . . . 3275
C.62.3 Source code . . . 3288
C.63 wc_mldsa.h . . . 3290
C.63.1 Functions . . . 3290
C.63.2 Functions Documentation . . . 3295
C.63.3 Source code . . . 3316
C.64 wc_mlkem.h . . . 3319
C.64.1 Functions . . . 3319

C.64.2 Functions Documentation . . . 3322
C.64.3 Source code . . . 3334
C.65 wc_port.h . . . 3335
C.65.1 Functions . . . 3335
C.65.2 Functions Documentation . . . 3337
C.65.3 Source code . . . 3351
C.66 wc_she.h . . . 3353
C.66.1 Functions . . . 3353
C.66.2 Functions Documentation . . . 3356
C.66.3 Source code . . . 3373
C.67 wc_slhdsa.h . . . 3375
C.67.1 Functions . . . 3375
C.67.2 Functions Documentation . . . 3380
C.67.3 Source code . . . 3402
C.68 wc_xmss.h . . . 3404
C.68.1 Functions . . . 3404
C.68.2 Functions Documentation . . . 3407
C.68.3 Source code . . . 3418
C.69 wolfio.h . . . 3419
C.69.1 Functions . . . 3419
C.69.2 Functions Documentation . . . 3424
C.69.3 Source code . . . 3454

D SSL/TLS Overview 3458
D.1 General Architecture . . . 3458
D.2 SSL Handshake . . . 3458
D.3 Differences between SSL and TLS Protocol Versions . . . 3458
D.3.1 SSL 3.0 . . . 3460
D.3.2 TLS 1.0 . . . 3460
D.3.3 TLS 1.1 . . . 3460
D.3.4 TLS 1.2 . . . 3460
D.3.5 TLS 1.3 . . . 3461

E RFCs, Specifications, and Reference 3462
E.1 Protocols . . . 3462
E.2 Stream Ciphers . . . 3462
E.3 Block Ciphers . . . 3462
E.4 Hashing Functions . . . 3462
E.5 Public Key Cryptography . . . 3462
E.6 Other . . . 3462

F Error Codes 3463
F.1 wolfSSL Error Codes . . . 3463
F.2 wolfCrypt Error Codes . . . 3466
F.3 Common Error Codes and their Solution . . . 3469
F.3.1 `ASN_NO_SIGNER_E` (-188) . . . 3469
F.3.2 `WANT_READ` (-323) . . . 3469

G Supported Cipher Suites, Key Exchange Groups, and Signature Algorithms 3470
G.1 A. Cipher Suites . . . 3470
G.1.1 A-1. TLS 1.3 . . . 3470
G.1.2 A-2. TLS 1.2 ECDHE-ECDSA . . . 3470
G.1.3 A-3. TLS 1.2 ECDHE-RSA . . . 3471
G.1.4 A-4. TLS 1.2 DHE-RSA . . . 3471
G.1.5 A-5. TLS 1.2 RSA (Static Key Exchange) . . . 3471

G.1.6 A-6. TLS 1.2 Static ECDH . . . 3472
G.1.7 A-7. TLS 1.2 PSK . . . 3472
G.1.8 A-8. TLS 1.2 DHE-PSK . . . 3473
G.1.9 A-9. TLS 1.2 ECDHE-PSK . . . 3473
G.1.10 A-10. TLS 1.2 DH_anon . . . 3474
G.2 B. Key Exchange Groups . . . 3474
G.2.1 B-1. NIST Elliptic Curves . . . 3474
G.2.2 B-2. Koblitz Curves . . . 3474
G.2.3 B-3. Brainpool Curves . . . 3474
G.2.4 B-4. X25519 / X448 . . . 3475
G.2.5 B-5. FFDHE . . . 3475
G.2.6 B-6. SM2 . . . 3475
G.2.7 B-7. ML-KEM Standalone . . . 3475
G.2.8 B-8. ML-KEM Hybrid (Standard) . . . 3475
G.2.9 B-9. ML-KEM Hybrid (Experimental) . . . 3476
G.2.10 B-10. Kyber Round 3 Legacy . . . 3476
G.3 C. Signature Algorithms . . . 3476
G.3.1 C-1. ECDSA (NIST Curves) . . . 3476
G.3.2 C-2. ECDSA (Brainpool, TLS 1.3 Only) . . . 3477
G.3.3 C-3. EdDSA . . . 3477
G.3.4 C-4. RSA-PSS RSAE . . . 3477
G.3.5 C-5. RSA-PSS PSS (TLS 1.3 Only) . . . 3477
G.3.6 C-6. RSA PKCS#1 v1.5 . . . 3477
G.3.7 C-7. SM2 . . . 3478
G.3.8 C-8. ML-DSA / Dilithium . . . 3478
G.4 D. configure Option #define Reference . . . 3478

H wolfSSL Hardware Optimization Support Reference 3480
H.1 A. Architecture-Level ISA/SIMD Optimizations . . . 3480
H.2 B. SP Math Library (Public Key Cryptography) . . . 3481
H.2.1 sp_int.c Inline Assembly Speedups . . . 3481
H.2.2 SP Key Size Configuration . . . 3481
H.3 C. Microcontroller/SoC Chip-Specific Hardware Accelerators . . . 3481
H.4 D. External Accelerators and Coprocessors . . . 3483
H.5 E. OS Kernel Hardware Offload Interfaces . . . 3483

I Experimenting with Post-Quantum Cryptography 3484
I.1 A Gentle Introduction to Post-Quantum Cryptography . . . 3484
I.1.1 Why Post-Quantum Cryptography? . . . 3484
I.1.2 How do we Protect Ourselves? . . . 3484
I.2 Getting Started with Post-Quantum algorithms in wolfSSL . . . 3485
I.2.1 Build Instructions . . . 3486
I.2.2 Making a Quantum Safe TLS Connection . . . 3486
I.3 Post Quantum Algorithm Variant Names . . . 3486
I.4 Cryptographic Artifact Sizes . . . 3487
I.5 Statistics . . . 3487
I.5.1 Runtime Binary Sizes . . . 3487
I.5.2 TLS 1.3 Data Transmission Sizes . . . 3487
I.5.3 Heap and Stack Usage . . . 3488
I.5.4 Benchmarks . . . 3493
I.6 Documentation . . . 3501
I.7 Post-Quantum Stateful Hash-Based Signatures . . . 3501
I.7.1 Motivation . . . 3501
I.7.2 LMS/HSS signatures . . . 3502

COPYRIGHT ©2024 wolfSSL Inc.

I.7.3 XMSS/XMSS^MT signatures . . . 3505
I.7.4 Developer Notes . . . 3510

J wolfSSL Porting Guide 3511
J.1 Purpose . . . 3511
J.2 Audience . . . 3511
J.3 Introduction . . . 3511
J.4 Porting wolfSSL . . . 3511
J.4.1 Data Types . . . 3511
J.4.2 Endianness . . . 3512
J.4.3 writev . . . 3512
J.4.4 Input / Output . . . 3513
J.4.5 Filesystem . . . 3513
J.4.6 Threading . . . 3514
J.4.7 Random Seed . . . 3514
J.4.8 Memory . . . 3515
J.4.9 Time . . . 3515
J.4.10 C Standard Library . . . 3515
J.4.11 Logging . . . 3516
J.4.12 Public Key Operations . . . 3516
J.4.13 Atomic Record Layer Processing . . . 3516
J.4.14 Features . . . 3516
J.5 Next Steps . . . 3517
J.5.1 wolfCrypt Test Application . . . 3517
J.6 Support . . . 3517

K wolfSM (ShangMi) 3518
K.1 Getting and Installing wolfSM . . . 3518
K.1.1 Get wolfSM from GitHub . . . 3518
K.1.2 Get wolfSSL from GitHub . . . 3518
K.1.3 Install SM code into wolfSSL . . . 3518
K.2 Building wolfSM . . . 3518
K.2.1 Optimized SM2 . . . 3519
K.3 Testing wolfSM . . . 3519
K.3.1 Testing TLS . . . 3519

1 Introduction

This manual is written as a technical guide to the wolfSSL embedded SSL/TLS library. It will explain how to build and get started with wolfSSL, provide an overview of build options, features, portability enhancements, support, and much more.

You can find the PDF version of this document here.

1.1 Why Choose wolfSSL?

There are many reasons to choose wolfSSL as your embedded SSL solution. Some of the top reasons include size (typical footprint sizes range from 20-100 kB), support for the newest standards (SSL 3.0, TLS 1.0, TLS 1.1, TLS 1.2, TLS 1.3, DTLS 1.0, DTLS 1.2, and DTLS 1.3), current and progressive cipher support (including stream ciphers), multi- platform, royalty free, and an OpenSSL compatibility API to ease porting into existing applications which have previously used the OpenSSL package. For a complete feature list, see Features Overview.

COPYRIGHT ©2024 wolfSSL Inc.

2 Building wolfSSL

wolfSSL was written with portability in mind and should generally be easy to build on most systems. If you have difficulty building wolfSSL, please don't hesitate to seek support through our support forums (https://www.wolfssl.com/forums) or contact us directly at support@wolfssl.com.

This chapter explains how to build wolfSSL on Unix and Windows, and provides guidance for building wolfSSL in a non-standard environment. You will find the "getting started" guide in Chapter 3 and an SSL tutorial in Chapter 11.

When using the autoconf / automake system to build wolfSSL, wolfSSL uses a single Makefile to build all parts and examples of the library, which is both simpler and faster than using Makefiles recursively.

2.1 Getting wolfSSL Source Code

The most recent version of wolfSSL can be downloaded from the wolfSSL website as a ZIP file:

https://www.wolfssl.com/download

After downloading the ZIP file, unzip the file using the `unzip` command. To use native line endings, enable the `-a` modifier when using unzip. From the unzip man page, the `-a` modifier functionality is described:

> [...] The -a option causes files identified by zip as text files (those with the 't' label in zipinfo listings, rather than 'b') to be automatically extracted as such, converting line endings, end-of-file characters and the character set itself as necessary. [...]

NOTE: Beginning with the release of wolfSSL 2.0.0rc3, the directory structure of wolfSSL was changed as well as the standard install location. These changes were made to make it easier for open source projects to integrate wolfSSL. For more information on header and structure changes, please see Library Headers and Structure Usage.

2.2 Building on Unix-like Systems

When building wolfSSL on Linux, *BSD, OS X, Solaris, or other *nix-like systems, use the autoconf system. To build wolfSSL you only need to run two commands from the wolfSSL root directory, `./configure` and `make`.

The `./configure` script sets up the build environment and you can append any number of build options to `./configure`. For a list of available build options, please see Build Options or run the following command to see a list of possible options to pass to the `./configure` script:

```
./configure --help
```

Once `./configure` has successfully executed, to build wolfSSL, run:

```
make
```

To install wolfSSL run:

```
make install
```

You may need superuser privileges to install, in which case precede the command with sudo:

```
sudo make install
```

To test the build, run the testsuite program from the root wolfSSL directory:

```
./testsuite/testsuite.test
```

Alternatively you can use autoconf to run the testsuite as well as the standard wolfSSL API and crypto tests:

```
make test
```

Further details about expected output of the testsuite program can be found in the Testsuite section. If you want to build only the wolfSSL library and not the additional items (examples, testsuite, benchmark app, etc.), you can run the following command from the wolfSSL root directory:

```
make src/libwolfssl.la
```

2.3 Building on Windows

In addition to the instructions below, you can find instructions and tips for building wolfSSL with Visual Studio here.

2.3.1 VS 2008

Solutions are included for Visual Studio 2008 in the root directory of the install. For use with Visual Studio 2010 and later, the existing project files should be able to be converted during the import process.

Note: If importing to a newer version of VS you will be asked: "Do you want to overwrite the project and its imported property sheets?" You can avoid the following by selecting "No". Otherwise if you select "Yes", you will see warnings about `EDITANDCONTINUE` being ignored due to `SAFESEH` specification. You will need to right click on the testsuite, sslSniffer, server, echoserver, echoclient, and client individually and modify their Properties->Configuration Properties->Linker->Advanced (scroll all the way to the bottom in Advanced window). Locate "Image Has Safe Exception Handlers" and click the drop down arrow on the far right. Change this to No (`/SAFESEH:NO`) for each of the aforementioned. The other option is to disable `EDITANDCONTINUE` which, we have found to be useful for debugging purposes and is therefore not recommended for production software.

2.3.2 VS 2010

You will need to download Service Pack 1 to build wolfSSL solution once it has been updated. If VS reports a linker error, clean and rebuild the project; the linker error should be taken care of.

2.3.3 VS 2013 (64 bit solution)

You will need to download Service Pack 4 to build wolfSSL solution once it has been updated. If VS reports a linker error, clean and rebuild the project; the linker error should be taken care of.

To test each build, choose "Build All" from the Visual Studio menu and then run the testsuite program. To edit build options in the Visual Studio project, select your desired project (wolfssl, echoclient, echoserver, etc.) and browse to the "Properties" panel.

Note: After the wolfSSL v3.8.0 release the build preprocessor macros were moved to a centralized file located at `IDE/WIN/user_settings.h`. This file can also be found in the project. To add features such as ECC or ChaCha20/Poly1305, add `#defines` here such as `HAVE_ECC` or `HAVE_CHACHA` / `HAVE_POLY1305`.

2.3.4 Cygwin

If building wolfSSL for Windows on a Windows development machine, we recommend using the included Visual Studio project files to build wolfSSL. However, if Cygwin is required here is a short guide on how our team achieved a successful build:

1. Go to https://www.cygwin.com/install.html and download `setup-x86_64.exe`
2. Run `setup-x86_64.exe` and install however you choose. Click through the installation menus until you reach the "Select Packages" stage.

3. Click on the “+” icon to expand “All”
4. Now go to the “Archive” section and select “unzip” drop down, change “Skip” to 6.0-15 (or some other version).
5. Under “Devel” click “autoconf” drop down and change “Skip” to “10-1” (or some other version)
6. Under “Devel” click “automake” drop down and change “Skip” to “10-1” (or some other version)
7. Under “Devel” click the “gcc-core” drop down and change “Skip” to 7.4.0-1 (NOTE: wolfSSL has not tested GCC 9 or 10 and as they are fairly new does not recommend using them until they have had a bit more time to be fine-tuned for development).
8. Under “Devel” click the “git” drop down and change “Skip” to 2.29.0-1 (or some other version)
9. Under “Devel” click “libtool” drop down and change “Skip” to “2.4.6-5” (or some other version)
10. Under “Devel” click the “make” drop down and change “Skip” to 4.2.1-1 (or some other version)
11. Click “Next” and proceed through the rest of the installation.

The additional packages list should include:

- unzip
- autoconf
- automake
- gcc-core
- git
- libtool
- make

2.3.4.1 Post Install

Open a Cygwin terminal and clone wolfSSL:

```
git clone https://github.com/wolfssl/wolfssl.git
cd wolfssl
./autogen.sh
./configure
make
make check
```

2.4 Building with Various Vendor IDEs

Download the complete wolfSSL source code package from the wolfSSL website, extract it, and apply the following settings.

2.4.1 1. wolfSSL Library Build Project

- **Register source files to be compiled**
  Register all `*.c` files under `./src` and all `*.c` files under `./wolfcrypt/src`.
  Exclude the following files:

./src/ssl_.c, ./src/x509_.c, src/conf.c, src/bio.c wolfcrypt/src/misc.c, ./wolfcrypt/src/evp.c

- **Define configuration options**
  Store the configuration definitions in a header file named `user_settings.h`.

Refer to: `-wolfssl/examples/configs/README.md -wolfssl/examples/configs/user_settings_template.`

- **Register predefined macro names**
  When `WOLFSSL_USER_SETTINGS` is defined, the above `user_settings.h` file is included at build time.
- **Register include paths**
  Register the path to the wolfSSL source file root and the path to the above `user_settings.h`.

2.4.2 2. Application Executable Build Project

- Register the wolfSSL library generated in step 1 as a link target.
- Register the wolfSSL source file root path and the configuration option path (`user_settings.h`) defined in step 1 as include paths.
- Specify `"WOLFSSL_USER_SETTINGS"` as a predefined macro name.

2.5 Building in a non-standard environment

While not officially supported, we try to help users wishing to build wolfSSL in a non-standard environment, particularly with embedded and cross-compilation systems. Below are some notes on getting started with this.

1. The source and header files need to remain in the same directory structure as they are in the wolfSSL download package.
2. Some build systems will want to explicitly know where the wolfSSL header files are located, so you may need to specify that. They are located in the `<wolfssl_root>/wolfssl` directory. Typically, you can add the `<wolfssl_root>` directory to your include path to resolve header problems.
3. wolfSSL defaults to a little endian system unless the configure process detects big endian. Since users building in a non-standard environment aren't using the configure process, `BIG_ENDIAN_ORDER` will need to be defined if using a big endian system.
4. wolfSSL benefits speed-wise from having a 64-bit type available. The configure process determines if long or long long is 64 bits and if so sets up a define. So if `sizeof(long)` is 8 bytes on your system, define `SIZEOF_LONG 8`. If it isn't but `sizeof(long long)` is 8 bytes, then define `SIZEOF_LONG_LONG 8`.
5. Try to build the library, and let us know if you run into any problems. If you need help, contact us at facts@wolfssl.com.
6. Some defines that can modify the build are listed in the following sub-sections, below. For more verbose descriptions of many options, please see Build Options.

2.5.1 Building into Yocto Linux

wolfSSL also includes recipes for building wolfSSL on Yocto Linux and OpenEmbedded. These recipes are maintained within the meta-wolfSSL layer as a GitHub repository, here: https://github.com/wolfSSL/meta-wolfssl. Building wolfSSL on Yocto Linux will require Git and bitbake. The following steps list how to get some wolfSSL products (that recipes exist for) built on Yocto Linux.

1. **Cloning wolfSSL meta**

   This can be done through a git-clone command of the following URL: https://github.com/wolfSSL/meta-wolfssl

2. **Insert the "meta-wolfSSL" layer into the build's bblayers.conf**

   Within the `BBLAYERS` section, add the path to the location where meta-wolfssl was cloned into. Example:

   ```
   BBLAYERS ?= "... \
   /path/to/meta-wolfssl/ \
   ..."
   ```

3. **Build a wolfSSL product recipe**

   bitbake can be used to build one of the three following wolfSSL product recipes: *wolfssl*, *wolfssh*, and *wolfmqtt*. Simply pass one of those recipes into the bitbake command (example: `bitbake wolfssl`). This allows the user to personally confirm compilation succeeds without issues.

4. **Edit local.conf**

The final step is to edit the build's local.conf file, which allows desired libraries to be included with the image being built. Edit the IMAGE_INSTALL_append line to include the name of the desired recipe(s). An example of this is shown below:

```
IMAGE_INSTALL_append = "wolfssl wolfssh wolfmqtt"
```

Once the image has been built, wolfSSL's default location (or related products from recipes) will be the /usr/lib/ directory.

Additionally, wolfSSL can be customized when building into Yocto by using the enable and disable options listed in Build Options. This requires creating a .bbappend file and placing it within the wolfSSL application/recipe layer. The contents of this file should include a line specifying content to concatenate onto the EXTRA_OECONF variable. An example of this is shown below to enable TLS 1.3 support through the TLS 1.3 enable option:

```
EXTRA_OECONF += "--enable-tls13"
```

Further documentation on building into Yocto can be found in the meta-wolfssl README, located here: https://github.com/wolfSSL/meta-wolfssl/blob/master/README.md

2.5.2 Building with Atollic TrueSTUDIO

Versions of wolfSSL following 3.15.5 include a TrueSTUDIO project file that is used to build wolfSSL on ARM M4-Cortex devices. The TrueSTUDIO project file simplifies the process of building on STM32 devices, is free to download, and is createed by Atollic - a part of ST Microelectronics. To build the wolfSSL static library project file in TrueSTUDIO, it will require the user perform the following steps after opening TrueSTUDIO:

1. Import the project into the workspace (File > Import)
2. Build the project (Project > Build project)

The build then includes the settings located inside of user_settings.h at build-time. The default content of the user_settings.h file is minimal, and does not contain many features. Users are able to modify this file and add or remove features with options listed in the remainder of this chapter.

2.5.3 Building with IAR

The <wolfssl_root>/IDE/IAR-EWARM directory contains the following files:

1. Workspace: wolfssl.eww The workspace includes wolfSSL-Lib library and wolfCrypt-test, wolfCrypt-benchmark executable projects.
2. wolfSSL-Lib Project: lib/wolfSSL-lib.ewp generates full set library of wolfCrypt and wolfSSL functions.
3. Test suites Project: test/wolfCrypt-test.ewp generates test.out test suites executable
4. Benchmark Project: benchmark/wolfCrypt-benchmark.ewp generates benchmark.out benchmark executable

These projects have been set up to generic ARM Cortex-M MPUs. In order to generate project for specific target MPU, take following steps.

1. Default Setting: Default Target of the projects are set to Cortex-M3 Simulator. user_settings.h includes default options for the projects. You can build and download the to the simulator. Open Terminal I/O window, by "view"->"Terminal I/O", and start execution.
2. Project option settings: For each project, choose appropriate "Target" options.
3. For executable projects: Add "SystemInit" and "startup" for your MPU, choose your debug "Driver".

4. For benchmark project: Choose option for current_time function or write your own “current_time” benchmark timer with WOLFSSL_USER_CURRTIME option.
5. Build and download: Go to “Project->Make” and “Download and Debug” in Menu bar for EWARM build and download.

2.5.4 Building on OS X and iOS

2.5.4.1 XCODE The `<wolfssl_root>/IDE/XCODE` directory contains the following files:

1. `wolfssl.xcworkspace` – workspace with library and testsuite client
2. `wolfssl_testsuite.xcodeproj` – project to run the testsuite.
3. `wolfssl.xcodeproj` – project to build OS/x and iOS libraries for wolfSSL and/or wolfCrypt
4. `wolfssl-FIPS.xcodeproj` – project to build wolfSSL and wolfCrypt-FIPS if available
5. `user_settings.h` – custom library settings, which are shared across projects

The library will output as `libwolfssl_osx.a` or `libwolfssl_ios.a` depending on the target. It will also copy the wolfSSL/wolfCrypt (and the CyaSSL/CtaoCrypt compatibility) headers into an `include` directory located in `Build/Products/Debug` or `Build/Products/Release`.

For the library and testsuite to link properly the build location needs to be configured relative to the workspace.

1. File -> Workspace Settings (or Xcode -> Preferences -> Locations -> Locations)
2. Derived Data -> Advanced
3. Custom -> Relative to Workspace
4. Products -> Build/Products

These Xcode projects define the `WOLFSSL_USER_SETTINGS` preprocessor to enable the `user_settings.h` file for setting macros across multiple projects.

If needed the Xcode preprocessors can be modified with these steps:

1. Click on the Project in “Project Navigator”.
2. Click on the “Build Settings” tab.
3. Scroll down to the “Apple LLVM 6.0 - Preprocessing” section.
4. Open the disclosure for “Preprocessor Macros” and use the “+” and “-” buttons to modify. Remember to do this for both Debug and Release.

This project should build wolfSSL and wolfCrypt using the default settings.

2.5.5 Building with GCC ARM

In the `<wolfssl_root>/IDE/GCC-ARM` directory, you will find an example wolfSSL project for Cortex M series, but it can be adopted for other architectures.

1. Make sure you have `gcc-arm-none-eabi` installed.
2. Modify the `Makefile.common`:
   - Use correct toolchain path `TOOLCHAIN`.
   - Use correct architecture ‘ARCHFLAGS’. See GCC ARM Options `-mcpu=name`.
   - Confirm memory map in linker.ld matches your flash/ram or comment out `SRC_LD = -T./linker.ld` in Makefile.common.
3. Use `make` to build the static library (libwolfssl.a), wolfCrypt test/benchmark and wolfSSL TLS client targets as `.elf` and `.hex` in `/Build`.

2.5.5.1 Building with generic makefile cross-compile Example `Makefile.common` changes for Raspberry Pi with Cortex-A53:

1. In Makefile.common change `ARCHFLAGS` to `-mcpu=cortex-a53 -mthumb`.

2. Comment out SRC_LD, since custom memory map is not applicable.
3. Clear TOOLCHAIN, so it will use default gcc. Set TOOLCHAIN =
4. Comment out LDFLAGS += --specs=nano.specs and LDFLAGS += --specs=nosys.specs to nosys and nano.

2.5.5.2 Building with configure with cross-compile The configure script in the main project directory can perform a cross-compile build with the gcc-arm-none-eabi tools. Assuming the tools are installed in your executable path:

```
./configure \
  --host=arm-none-eabi \
  CC=arm-none-eabi-gcc \
  AR=arm-none-eabi-ar \
  STRIP=arm-none-eabi-strip \
  RANLIB=arm-none-eabi-ranlib \
  --prefix=/path/to/build/wolfssl-arm \
  CFLAGS="-march=armv8-a --specs=nosys.specs \
      -DHAVE_PK_CALLBACKS -DWOLFSSL_USER_IO -DWOLFSSL_NO_SOCK -DNO_WRITEV" \
  --disable-filesystem --enable-crypttests \
  --disable-shared
make
make install
```

If you are building for a 32-bit architecture, add -DTIME_T_NOT_64BIT to the list of CFLAGS.

2.5.6 Building on Keil MDK-ARM

You can find detailed instructions and tips for building wolfSSL on Keil MDK-ARM here.

Note: If MDK-ARM is not installed in the default installation location, you need to change all of the referencing path definitions in the project file to the install location.

2.6 Features Defined as C Pre-processor Macro

2.6.1 Removing Features

The following defines can be used to remove features from wolfSSL. This can be helpful if you are trying to reduce the overall library footprint size. In addition to defining a NO_<feature-name> define, you can also remove the respective source file as well from the build (but not the header file).

2.6.1.1 NO_WOLFSSL_CLIENT Removes calls specific to the client and is for server-only builds. You should only use this if you want to remove a few calls for the sake of size.

2.6.1.2 NO_WOLFSSL_SERVER Likewise removes calls specific to the server side.

2.6.1.3 NO_DES3 Removes the use of DES3 encryptions. DES3 is built-in by default because some older servers still use it and it's required by SSL 3.0. NO_DH and NO_AES are the same as the two above, they are widely used.

2.6.1.4 WOLFSSL_DES_ECB Enables DES-ECB (Electronic Codebook) mode. Off by default. Used by some legacy protocols.

2.6.1.5 WC_ASYNC_ENABLE_3DES Enables asynchronous 3DES operations via the wolfSSL async crypto framework.

2.6.1.6 FREESCALE_LTC_DES Enables Freescale/NXP LTC hardware DES/3DES acceleration.

2.6.1.7 NO_DSA Removes DSA since it's being phased out of popular use.

2.6.1.8 NO_ERROR_STRINGS Disables error strings. Error strings are located in `src/internal.c` for wolfSSL or `wolfcrypt/src/asn.c` for wolfCrypt.

2.6.1.9 NO_HMAC Removes HMAC from the build.

NOTE: SSL/TLS depends on HMAC but if you are only using wolfCrypt IE build option `WOLFCRYPT_ONLY` then HMAC can be disabled in this case.

2.6.1.10 HAVE_HKDF Enables HKDF (HMAC-based Extract-and-Expand Key Derivation Function) per RFC 5869. Used for TLS 1.3 key derivation and general-purpose key derivation from shared secrets.

2.6.1.11 WOLFSSL_HMAC_COPY_HASH Copies the hash state instead of re-initializing for each HMAC operation. Improves performance when the same key is used for multiple HMAC computations.

2.6.1.12 STM32_HMAC Enables STM32 hardware HMAC acceleration using the STM32 HASH peripheral.

2.6.1.13 WC_ASYNC_ENABLE_HMAC Enables asynchronous HMAC operations via the wolfSSL async crypto framework.

2.6.1.14 WOLFSSL_DEVCRYPTO_HMAC Enables HMAC acceleration via the Linux `/dev/crypto` interface.

2.6.1.15 WOLFSSL_KCAPI_HMAC Enables HMAC operations through the Linux kernel crypto API (AF_ALG).

2.6.1.16 NO_MD4 Removes MD4 from the build, MD4 is broken and shouldn't be used.

2.6.1.17 NO_MD5 Removes MD5 from the build.

2.6.1.18 HAVE_MD5_CUST_API Enables a custom MD5 API. Allows user-provided MD5 implementation.

2.6.1.19 STM32_NOMD5 Disables STM32 hardware MD5 acceleration. Use when STM32 crypto is enabled but MD5 should use the software implementation.

2.6.1.20 WC_ASYNC_ENABLE_MD5 Enables asynchronous MD5 operations via the wolfSSL async crypto framework.

2.6.1.21 NO_SHA Removes SHA-1 from the build.

2.6.1.22 NO_SHA256 Removes SHA-256 from the build.

2.6.1.23 NO_PSK Turns off the use of the pre-shared key extension. It is built-in by default.

2.6.1.24 NO_PWDBASED Disables password-based key derivation functions such as PBKDF1, PBKDF2, and PBKDF from PKCS #12.

2.6.1.25 HAVE_PKCS7 Enables PKCS#7 (Cryptographic Message Syntax) support for signed data, enveloped data, encrypted data, and compressed data content types.

2.6.1.26 NO_PKCS7_STREAM Disables PKCS#7 streaming mode. Streaming mode allows processing PKCS#7 data incrementally.

2.6.1.27 NO_PKCS7_ENCRYPTED_DATA Disables the PKCS#7 EncryptedData content type. Reduces code size when only SignedData or EnvelopedData are needed.

2.6.1.28 NO_PKCS7_COMPRESSED_DATA Disables the PKCS#7 CompressedData content type. When CompressedData support is enabled (i.e., this macro is not defined), zlib (HAVE_LIBZ) is required.

2.6.1.29 WC_PKCS7_STREAM_DEBUG Enables debug output for PKCS#7 streaming operations.

2.6.1.30 WOLFSSL_PKCS7_MAX_DECOMPRESSION Sets the maximum size for PKCS#7 decompression output. Prevents memory exhaustion from maliciously crafted compressed data.

2.6.1.31 HAVE_PKCS7_RSA_RAW_SIGN_CALLBACK Enables a custom callback for raw RSA signing in PKCS#7. Allows HSM or external signing integration.

2.6.1.32 HAVE_PKCS7_ECC_RAW_SIGN_CALLBACK Enables a custom callback for raw ECC signing in PKCS#7. Allows HSM or external signing integration.

2.6.1.33 HAVE_X963_KDF Enables ANSI X9.63 Key Derivation Function. Used for ECC-based key agreement in PKCS#7 EnvelopedData.

2.6.1.34 NO_RC4 Removes the use of the ARC4 stream cipher from the build. ARC4 is built-in by default because it is still popular and widely used.

2.6.1.35 NO_SESSION_CACHE Can be defined when a session cache is not needed. This should reduce memory use by nearly 3 kB.

2.6.1.36 NO_TLS Turns off TLS. We do not recommend turning off TLS.

2.6.1.37 SMALL_SESSION_CACHE Can be defined to limit the size of the SSL session cache used by wolfSSL. This will reduce the default session cache from 33 sessions to 6 sessions and save approximately 2.5 kB.

2.6.1.38 NO_RSA Removes support for the RSA algorithm.

2.6.1.39 WC_NO_RSA_OAEP Removes code for OAEP padding.

2.6.1.40 WOLFSSL_RSA_VERIFY_INLINE Enables inline RSA verify, returning a pointer into the input buffer rather than copying the output. Reduces memory usage for RSA verify operations.

2.6.1.41 WC_RSA_DIRECT Enables the direct RSA encrypt/decrypt API (`wc_RsaDirect`). Provides raw RSA operations without padding, useful for custom protocols.

2.6.1.42 WC_RSA_NO_PADDING Enables the no-padding RSA mode. Allows RSA operations without any padding scheme applied. Use with caution - typically only for custom implementations.

2.6.1.43 WOLFSSL_RSA_KEY_CHECK Enables RSA key pair consistency checking via `wc_CheckRsaKey()`. Validates that the public and private key components are mathematically consistent.

2.6.1.44 WOLFSSL_RSA_CHECK_D_ON_DECRYPT Validates the RSA private exponent d before each decrypt operation. Provides additional security against fault injection attacks at the cost of performance.

2.6.1.45 WOLFSSL_RSA_DECRYPT_TO_0_LEN Allows RSA decrypt operations to return a zero-length result (empty plaintext). By default, a zero-length decryption result is treated as an error.

2.6.1.46 NO_RSA_BOUNDS_CHECK Disables bounds checking on RSA input data. By default, wolfSSL validates that the input value is less than the RSA modulus.

2.6.1.47 SHOW_GEN Enables progress indicator (dots) during RSA key generation. Useful for user feedback during long key generation operations.

2.6.1.48 WOLFSSL_PSS_LONG_SALT Allows RSA-PSS signatures to use a salt length longer than the hash output length. Some implementations use salt length equal to the key size minus overhead.

2.6.1.49 WOLFSSL_PSS_SALT_LEN_DISCOVER Enables automatic discovery of the PSS salt length during RSA-PSS signature verification. Tries different salt lengths to find a match.

2.6.1.50 WC_RSA_NONBLOCK_TIME Enables time-based non-blocking RSA operations. Allows RSA operations to yield after a configurable time period. Requires `WC_RSA_NONBLOCK`.

2.6.1.51 WOLFSSL_MP_INVMOD_CONSTANT_TIME Uses constant-time modular inverse computation. Protects against timing side-channel attacks during RSA private key operations.

2.6.1.52 WC_RSA_NO_FERMAT_CHECK Disables the Fermat factorization proximity check during RSA key generation. By default, wolfSSL verifies that p and q are not too close together, which would make the key vulnerable to Fermat's factorization method.

2.6.1.53 FP_MAX_BITS Sets the maximum key size in bits when using fast math (`USE_FAST_MATH`). The value should be set to key size times 2. For example, for RSA 3072 set to 6144. Default is 4096 (supporting up to RSA 2048).

COPYRIGHT ©2024 wolfSSL Inc.

2.6.1.54 WOLFSSL_HAVE_SP_RSA Enables Single Precision (SP) math optimizations for RSA operations. SP math provides significant performance improvements for common key sizes (2048, 3072, 4096).

2.6.1.55 WOLFSSL_SP_ASM Enables assembly-optimized SP math routines. Provides maximum performance for RSA, ECC, and DH operations on supported platforms (x86_64, ARM, RISC-V).

2.6.1.56 WC_ASYNC_ENABLE_RSA Enables asynchronous RSA operations via the wolfSSL async crypto framework. Allows RSA operations to be offloaded to hardware accelerators.

2.6.1.57 WOLFSSL_KCAPI_RSA Enables RSA operations through the Linux kernel crypto API (AF_ALG). Offloads RSA to the kernel's crypto subsystem.

2.6.1.58 WOLFSSL_AFALG_XILINX_RSA Enables RSA acceleration through AF_ALG on Xilinx platforms using Xilinx crypto hardware.

2.6.1.59 WOLFSSL_SE050_NO_RSA Disables RSA through the NXP SE050 secure element. Other SE050 operations remain available.

2.6.1.60 WOLFSSL_XILINX_CRYPT Enables Xilinx hardware crypto acceleration for RSA and other algorithms on Xilinx FPGA/SoC platforms.

2.6.1.61 NO_AES_CBC Turns off AES-CBC algorithm support.

2.6.1.62 NO_AES_DECRYPT Can be set to reduce code size. Set to disable the decrypt and only support encryption.

2.6.1.63 WOLFCRYPT_ONLY Enables wolfCrypt only while disabling TLS.

2.6.1.64 NO_CAMELLIA_CBC Disables Camellia CBC support but only applies to TLS cipher suites only.

2.6.1.65 NO_AES Disables AES algorithm support.

2.6.1.66 NO_AES_128 Used for AES key size selection at compile time.

2.6.1.67 NO_AES_192 Used for AES key size selection at compile time.

2.6.1.68 NO_AES_256 Used for AES key size selection at compile time.

2.6.1.69 NO_AESGCM_AEAD Used for disabliing TLS cipher suites that use AES GCM. It is used internally when no AES GCM cipher suites are enabled, but can also be used to limit cipher suites.

2.6.1.70 NO_ASN_TIME Disables time checking for ASN. Note: This should be used with caution because all certificate begin/end date checking will be skipped.

2.6.1.71 NO_BIO Disables the BIO (Basic I/O) abstraction layer. Reduces code size when BIO functionality such as `BIO_new()`, `BIO_read()`, `BIO_write()` is not needed.

2.6.1.72 WOLFSSL_CERT_PIV Enables PIV (Personal Identity Verification) certificate support for government smart card applications.

2.6.1.73 WOLFSSL_CERT_GEN_CACHE Caches the DER encoding during certificate generation for reuse without re-encoding.

2.6.1.74 WOLFSSL_CERT_SIGN_CB Enables a callback function for certificate signing, allowing external signing operations (e.g., HSM).

2.6.1.75 WOLFSSL_CERT_NAME_ALL Stores all certificate name components including initials, given name, and DN qualifier. Uses more memory but provides full name access.

2.6.1.76 WOLFSSL_MULTI_ATTRIB Enables multi-valued Relative Distinguished Name (RDN) attributes in certificates.

2.6.1.77 WOLFSSL_DER_TO_PEM Enables DER to PEM format conversion functions.

2.6.1.78 WOLFSSL_PUB_PEM_TO_DER Enables public key PEM to DER format conversion.

2.6.1.79 WOLFSSL_KEY_TO_DER Enables key to DER encoding functions for RSA and ECC private keys.

2.6.1.80 ASN_BER_TO_DER Enables BER (Basic Encoding Rules) to DER (Distinguished Encoding Rules) conversion. Required for parsing BER-encoded certificates and CMS/PKCS#7 data.

2.6.1.81 WOLFSSL_DUP_CERTPOL Allows duplicate certificate policy OIDs in the certificate policies extension. By default, duplicates cause a validation error.

2.6.1.82 NO_VERIFY_OID Disables OID verification during certificate parsing.

2.6.1.83 NO_SKID Disables Subject Key Identifier extension processing.

2.6.1.84 NO_STRICT_ECDSA_LEN Relaxes strict ECDSA signature length checking. Allows non-minimal DER encoding of ECDSA signatures for interoperability.

2.6.1.85 ALLOW_SELFSIGNED_INVALID_CERTSIGN Allows self-signed certificates that do not have the keyCertSign bit set in the Key Usage extension.

2.6.1.86 ALLOW_V1_EXTENSIONS Allows X.509 v1 certificates to contain extensions. Per RFC 5280, extensions are only valid in v3 certificates.

2.6.1.87 USE_WOLF_VALIDDATE Uses wolfSSL's own date validation implementation instead of the platform's.

2.6.1.88 USE_WOLF_TM Uses wolfSSL's own `struct tm` definition instead of the system-provided one. Needed on platforms without a standard `struct tm`.

2.6.1.89 WC_ASN_RUNTIME_DATE_CHECK_CONTROL Enables runtime control of certificate date checking. Allows enabling or disabling date validation at runtime via API.

2.6.1.90 WOLFSSL_AFTER_DATE_CLOCK_SKEW Sets the clock skew tolerance (in seconds) for certificate not-after date checking. Allows certificates to be valid slightly past their expiration.

2.6.1.91 WOLFSSL_BEFORE_DATE_CLOCK_SKEW Sets the clock skew tolerance (in seconds) for certificate not-before date checking. Allows certificates to be accepted slightly before their validity period.

2.6.1.92 NO_WOLFSSL_SKIP_TRAILING_PAD Disables skipping of trailing padding bytes in ASN.1 parsing.

2.6.1.93 NO_WOLFSSL_STUB Disables stub implementations of unimplemented OpenSSL compatibility functions. Without stubs, calling unimplemented functions will result in a linker error rather than a runtime failure.

2.6.1.94 WOLFSSL_ALT_NAMES Enables Subject Alternative Name (SAN) support in certificate generation and parsing.

2.6.1.95 WOLFSSL_ALT_NAMES_NO_REV Stores Subject Alternative Names without reversing the order. By default, SANs are stored in reverse order.

2.6.1.96 WOLFSSL_ALTERNATIVE_DOWNGRADE Uses an alternative protocol downgrade detection mechanism instead of the standard approach.

2.6.1.97 WOLFSSL_IP_ALT_NAME Enables IP address entries in Subject Alternative Names.

2.6.1.98 WOLFSSL_JNI Enables APIs and behaviors needed for Java JNI (wolfSSL JNI/JSSE) compatibility.

2.6.1.99 WOLFSSL_RID_ALT_NAME Enables Registered ID entries in Subject Alternative Names.

2.6.1.100 WOLFSSL_EKU_OID Enables Extended Key Usage OID support for additional OIDs beyond the standard set.

2.6.1.101 WOLFSSL_ACERT Enables X.509 attribute certificate support (RFC 5755).

2.6.1.102 IGNORE_KEY_EXTENSIONS Ignores key usage and extended key usage extensions during certificate validation.

2.6.1.103 WOLFSSL_ALLOW_CRIT_AIA Allows the Authority Information Access (AIA) extension to be marked as critical. By default, a critical AIA causes a validation error.

2.6.1.104 WOLFSSL_ALLOW_CRIT_AKID Allows the Authority Key Identifier extension to be marked as critical.

2.6.1.105 WOLFSSL_ALLOW_CRIT_SKID Allows the Subject Key Identifier extension to be marked as critical.

2.6.1.106 WOLFSSL_ALLOW_MAX_FRAGMENT_ADJUST Allows runtime adjustment of the maximum fragment size after the initial TLS negotiation has completed.

2.6.1.107 WOLFSSL_ALLOW_NO_SUITES Allows creation of SSL/CTX objects even when no cipher suites are available. Normally an error is raised if no suites match the build configuration.

2.6.1.108 WOLFSSL_ALLOW_NO_CN_IN_SAN Allows certificates that have a Subject Alternative Name (SAN) extension but no Common Name (CN) in the subject.

2.6.1.109 WC_ASN_UNKNOWN_EXT_CB Enables a callback for handling unknown certificate extensions. The callback receives the OID and extension data for custom processing.

2.6.1.110 WOLFSSL_ASN_ALL Enables all optional ASN.1 features at once.

2.6.1.111 WOLFSSL_ASN_CA_ISSUER Enables parsing of the CA Issuer field in the Authority Information Access extension.

2.6.1.112 WOLFSSL_ASN_PRINT Enables ASN.1 structure printing functions for debugging and inspection of DER-encoded data.

2.6.1.113 WOLFSSL_ASN_INT_LEAD_0_ANY Allows any leading zero byte in ASN.1 INTEGER encoding. By default, only minimal encoding is accepted.

2.6.1.114 WOLFSSL_ASN_PARSE_KEYUSAGE Enables parsing of the Key Usage extension during certificate processing.

2.6.1.115 WOLFSSL_ASN_TIME_STRING Enables conversion of ASN.1 time values to human-readable string format.

2.6.1.116 ASN_TEMPLATE_SKIP_ISCA_CHECK Skips the isCA check in ASN.1 template-based certificate parsing.

2.6.1.117 HAVE_OID_ENCODING Enables OID (Object Identifier) encoding support for generating ASN.1 OIDs from dotted-decimal notation.

2.6.1.118 WOLFSSL_OLD_OID_SUM Uses the old OID sum calculation method. For backwards compatibility with applications that depend on specific OID sum values.

2.6.1.119 WOLFSSL_OPENVPN Enables OpenVPN compatibility behaviors in wolfSSL. Required when using wolfSSL as the TLS provider for OpenVPN.

2.6.1.120 HAVE_OCSP_RESPONDER Enables OCSP responder functionality. Allows wolfSSL to act as an OCSP responder that signs and sends OCSP responses.

2.6.1.121 WOLFSSL_OCSP_PARSE_STATUS Enables parsing of OCSP response status fields for detailed status information.

2.6.1.122 HAVE_PKCS8 Enables PKCS#8 private key format support for importing and exporting encrypted and unencrypted private keys.

2.6.1.123 HAVE_PKCS12 Enables PKCS#12 (PFX) format support for bundling private keys, certificates, and CA chains into a single encrypted file.

2.6.1.124 WOLFSSL_DILITHIUM_NO_ASN1 Disables ASN.1 encoding/decoding for Dilithium keys and signatures. Uses raw format instead.

2.6.1.125 WOLFSSL_DISABLE_EARLY_SANITY_CHECKS Disables early sanity checks on incoming TLS messages. May be needed for interoperability with non-compliant TLS implementations.

2.6.1.126 WOLFSSL_DILITHIUM_FIPS204_DRAFT Enables FIPS 204 draft version of Dilithium parameters.

2.6.1.127 HAVE_SPHINCS Enables SPHINCS+ post-quantum signature scheme support.

2.6.1.128 WC_ENABLE_ASYM_KEY_IMPORT Enables generic asymmetric key import functions for Ed25519, Ed448, Curve25519, and Curve448.

2.6.1.129 WC_ENABLE_ASYM_KEY_EXPORT Enables generic asymmetric key export functions for Ed25519, Ed448, Curve25519, and Curve448.

2.6.1.130 WOLFSSL_X509_NAME_AVAILABLE Enables the X509_NAME API for OpenSSL-compatible certificate name access.

2.6.1.131 WOLFSSL_HAVE_ISSUER_NAMES Stores pointers to issuer name components, their lengths and encodings for efficient access.

2.6.1.132 WOLFSSL_ASN_KEY_SIZE_ENUM Uses an enum for AES key size representation in ASN.1 processing instead of raw integers.

2.6.1.133 HAVE_SMIME Enables S/MIME (Secure/Multipurpose Internet Mail Extensions) support for email signing and encryption.

2.6.1.134 WC_RC2 Enables RC2 cipher support. Primarily needed for legacy PKCS#12 file compatibility.

2.6.1.135 WOLFSSL_MD2 Enables MD2 hash algorithm support. Only needed for legacy certificate compatibility. Not recommended for new applications.

2.6.1.136 NO_CHECK_PRIVATE_KEY This macro disables additional private key checking that is on by default. This enables checking to validate the private key is a pair for the public key. It is supported for RSA, ECDSA, ED25519, ED448, Falcon, Dilithium and Sphincs.

2.6.1.137 NO_CIPHER_SUITE_ALIASES Disables cipher suite name aliases. Only the primary cipher suite name will be recognized, not alternative names.

2.6.1.138 NO_CHAPOL_AEAD Disables ChaCha20-Poly1305 AEAD cipher suites. Use when ChaCha20-Poly1305 support is not desired even though the algorithms are compiled in.

2.6.1.139 NO_DH Disables Diffie-Hellman (DH) support.

2.6.1.140 NO_ED25519_CLIENT_AUTH Disables TLS client authentication support for ED25519. It is used to reduce memory usage during TLS if ED25519 is not used, since it requires caching messages.

2.6.1.141 NO_ED448_CLIENT_AUTH Disables client authentication for ED448.

2.6.1.142 NO_FORCE_SCR_SAME_SUITE By default secure renegotiation requires using the same cipher suite. This disables that requirement.

2.6.1.143 NO_MULTIBYTE_PRINT Used for compiling out special characters that embedded devices may have problems with.

2.6.1.144 NO_OLD_SSL_NAMES This disables some of the old OpenSSL compatibility macros for using wolfSSL and OpenSSL together.

2.6.1.145 NO_OLD_WC_NAMES Removes unneeded namespace.

2.6.1.146 NO_OLD_POLY1305 This disables support for the old ChaCha20/Poly1305 TLS 1.2 cipher suite that is typically used for interop.

2.6.1.147 NO_HANDSHAKE_DONE_CB Disables support for the handshake callback set with `wolfSSL_SetHsDoneCb`. This option is useful for reducing code size.

2.6.1.148 NO_STDIO_FILESYSTEM This disables include of stdio.h. Used with portability.

2.6.1.149 NO_TLS_DH Excludes TLS DH. Should not negotiate cipher suites based on ephemeral finite-field Diffie-Hellman key agreement.

2.6.1.150 NO_WOLFSSL_CM_VERIFY Disables the Certificate Manager verify callback. The verify callback allows intercepting errors and overriding them. This option is useful for reducing code size.

2.6.1.151 NO_WOLFSSL_DIR Disable directory support.

2.6.1.152 NO_WOLFSSL_RENESAS_TSIP_TLS_SESSION For disabling only the TSIP TLS-linked Common key encryption method. Note: This is a Renesas RX TSIP specific define.

2.6.1.153 NO_WOLFSSL_SHA256 This applies to TLS 1.3 only. It allows SHA2-256 to be enabled and usable from wolfCrypt, but exclude it from TLS 1.3.

2.6.1.154 WOLFSSL_BLIND_PRIVATE_KEY Used as a mask to blind the private key. The blinding is used to protect against Rowhammer attacks.

2.6.1.155 WOLFSSL_DTLS13_NO_HRR_ON_RESUME If defined, a DTLS server will not do a cookie exchange on successful client resumption: the resumption will be faster (one RTT less) and will consume less bandwidth (one ClientHello and one HelloVerifyRequest/HelloRetryRequest less). On the other hand, if a valid SessionID/ticket/psk is collected, forged clientHello messages will consume resources on the server. For DTLS 1.3, using this option also allows for the server to process Early Data/0-RTT Data. Without this, the Early Data would be dropped since the server doesn't enter stateful processing until receiving a verified ClientHello with the cookie. To allow DTLS 1.3 resumption without the cookie exchange:- Compile wolfSSL with WOLFSSL_DTLS13_NO_HRR_ON_RESUME defined - Call wolfSSL_dtls13_no_hrr_on_resume(ssl, 1) on the WOLFSSL object to disable the cookie exchange on resumption - Continue like with a normal connection.

2.6.1.156 WOLFSSL_DTLS13_SEND_MOREACK_DEFAULT Enables sending more ACK messages by default in DTLS 1.3 for improved reliability on lossy networks.

2.6.1.157 WOLFSSL_NO_CLIENT_AUTH Disables the caching code required for using Ed25519 and Ed448.

2.6.1.158 WOLFSSL_NO_CURRDIR Portability macro for platforms that do not support ./ for test paths in wolfssl/test.h. Applies to testing tools only.

2.6.1.159 WOLFSSL_NO_DEF_TICKET_ENC_CB No default ticket encryption callback, server only. The application must set its own callback to use session tickets.

2.6.1.160 WOLFSSL_NO_DTLS_SIZE_CHECK Disables DTLS record size validation checks. May be needed for interoperability with implementations that send non-standard record sizes.

2.6.1.161 WOLFSSL_NO_ETM_ALERT Suppresses the alert message when Encrypt-Then-MAC extension negotiation fails. Silently falls back to MAC-then-encrypt.

2.6.1.162 WOLFSSL_NO_SOCK Portability macro for disabling built-in socket support. If using TLS without sockets typically WOLFSSL_USER_IO would be defined and callbacks used for send/recv.

2.6.1.163 WOLFSSL_NO_STRICT_CIPHER_SUITE Relaxes strict cipher suite validation requirements. Allows cipher suites that may not perfectly match the negotiated protocol version.

2.6.1.164 WOLFSSL_NO_TICKET_EXPIRE Disables session ticket expiration checking. Session tickets will be accepted regardless of their age.

2.6.1.165 WOLFSSL_NO_TLS12 Define to exclude TLS 1.2.

2.6.1.166 WOLFSSL_PEM_TO_DER Key and cert generation feature support for disabling PEM to DER.

2.6.1.167 WOLFSSL_NO_SIGALG Disables the signature algorithms extension

2.6.1.168 NO_RESUME_SUITE_CHECK Disables the check of cipher suite when resuming a TLS connection

2.6.1.169 NO_ASN Removes support for ASN formatted certificate processing.

2.6.1.170 NO_OLD_TLS Removes support for SSLv3, TLSv1.0 and TLSv1.1. This macro is defined by default; it is only left undefined when old TLS support is explicitly requested (for example with `--enable-oldtls`, `--enable-tlsv10` or `--enable-sslv3`).

Warning: Leaving `NO_OLD_TLS` undefined produces a build that is **not conformant** with RFC 9846, RFC 8996 or RFC 9325. RFC 8996 deprecates TLS 1.0 and TLS 1.1, and RFC 9846 states that SSL 2.0, SSL 3.0, TLS 1.0 and TLS 1.1 "MUST NOT be negotiated for any reason". Only build without `NO_OLD_TLS` when legacy interoperability requires it, and do not claim conformance with those RFCs for such a build.

2.6.1.171 WOLFSSL_AEAD_ONLY Removes support for non-AEAD algorithms. AEAD stands for "authenticated encryption with associated data" which means these algorithms (such as AES-GCM) do not just encrypt and decrypt data, they also assure confidentiality and authenticity of that data.

2.6.1.172 WOLFSSL_SP_NO_2048 Removes RSA/DH 2048-bit Single-Precision (SP) optimization.

2.6.1.173 WOLFSSL_SP_NO_3072 Removes RSA/DH 3072-bit Single-Precision (SP) optimization.

2.6.1.174 WOLFSSL_SP_NO_256 Removes ECC Single-Precision (SP) optimization for SECP256R1. Only applies to `WOLFSSL_SP_MATH`.

2.6.2 Enabling Features macros (on by default)

2.6.2.1 HAVE_TLS_EXTENSIONS Enables support for TLS extensions, which are required for most TLS builds. Enabled by default with `./configure`, but needs to be manually defined if building with `WOLFSSL_USER_SETTINGS`.

2.6.2.2 HAVE_SUPPORTED_CURVES Enables the TLS supported curves and key share extensions used with TLS. Required with ECC, Curve25519 and Curve448. Enabled by default with `./configure`, but needs to be manually defined if building with `WOLFSSL_USER_SETTINGS`.

2.6.2.3 HAVE_EXTENDED_MASTER Enables extended master secret PRF for calculation of session keys used with TLS v1.2 and older. The PRF method is on by default and is considered more secure. This is on by default if using `./configure`, but needs to be manually defined if building with `WOLFSSL_USER_SETTINGS`.

2.6.2.4 HAVE_ENCRYPT_THEN_MAC Enables encrypt-then-mac support to perform mac after encryption with block ciphers. This is the default and improves security. This is on by default if using `./configure`, but needs to be manually defined if building with `WOLFSSL_USER_SETTINGS`.

2.6.2.5 HAVE_ONE_TIME_AUTH Required if using Chacha20/Poly1305 with TLS v1.2 for setting up Poly authentication. This is on by default with ChaCha20/Poly1305 if using `./configure`, but needs to be manually defined if building with `WOLFSSL_USER_SETTINGS`.

2.6.2.6 WOLFSSL_ASN_TEMPLATE Enables the new version of ASN parsing code that uses template-based ASN.1 processing. This parsing adheres to standard ASN.1 rules and uses a template structure to dictate encoding and decoding, allowing the parser code to generalize across templates. This is on by default if using `./configure`, but needs to be manually defined if building with `WOLFSSL_USER_SETTINGS`.

2.6.2.7 WOLFSSL_DEBUG_ASN_TEMPLATE Enables debugging output when using ASN.1 templates. Only relevant when used with `WOLFSSL_ASN_TEMPLATE`.

2.6.2.8 WOLFSSL_DEBUG_CERTS Enables debug logging for certificate processing operations including parsing, validation, and chain building.

2.6.2.9 WOLFSSL_ASN_TEMPLATE_TYPE_CHECK Use ASN functions to better test compiler type issues for testing. Only relevant when used with `WOLFSSL_ASN_TEMPLATE`

2.6.3 Enabling Features Disabled by Default

2.6.3.1 WOLFSSL_CERT_GEN Turns on wolfSSL's certificate generation functionality. See Keys and Certificates for more information.

2.6.3.2 WOLFSSL_DER_LOAD Allows loading DER-formatted CA certs into the wolfSSL context (`WOLFSSL_CTX`) using the function `wolfSSL_CTX_der_load_verify_locations()`.

2.6.3.3 WOLFSSL_DTLS Turns on the use of DTLS, or datagram TLS. This isn't widely supported or used.

2.6.3.4 WOLFSSL_DTLS_CID Enables DTLS Connection ID support (RFC 9146). Allows DTLS connections to survive IP address changes by identifying connections with a CID rather than the transport address.

2.6.3.5 WOLFSSL_DTLS_DROP_STATS Enables tracking of DTLS packet drop statistics for monitoring and debugging DTLS connection quality.

2.6.3.6 WOLFSSL_DTLS_DISALLOW_FUTURE Rejects DTLS records that have a future epoch number. Provides stricter epoch validation during DTLS communication.

2.6.3.7 WOLFSSL_ALLOW_TLSV10 Allows TLS 1.0 connections. TLS 1.0 is disabled by default for security reasons. Only enable when legacy compatibility is required. Implies that `NO_OLD_TLS` is not defined.

Warning: Defining `WOLFSSL_ALLOW_TLSV10` produces a build that is **not conformant** with RFC 9846, RFC 8996 or RFC 9325. TLS 1.0 is deprecated by RFC 8996, and RFC 9846 states it "MUST NOT be negotiated for any reason".

2.6.3.8 WOLFSSL_ALLOW_SSLV3 Allows SSL 3.0 connections. SSL 3.0 is disabled by default for security reasons. Only enable when legacy compatibility is required. Implies that `NO_OLD_TLS` is not defined.

Warning: Building with `--enable-sslv3` violates RFC 7568. SSL 3.0 is deprecated by RFC 7568 and states "SSLv3 MUST NOT be used."

2.6.3.9 WOLFSSL_ALLOW_TLS_SHA1 Allows SHA-1 based cipher suites and signatures in TLS, even when SHA-1 is otherwise restricted by security policy.

2.6.3.10 WOLFSSL_EITHER_SIDE Allows the same `WOLFSSL_CTX` to be used for both client and server connections. By default, a context is configured for either client or server at creation time.

2.6.3.11 WOLFSSL_EGD_NBLOCK Enables non-blocking EGD (Entropy Gathering Daemon) support for random number generation on systems using EGD.

2.6.3.12 HAVE_SNI Enables Server Name Indication (SNI) TLS extension support (RFC 6066). Allows clients to indicate which hostname they are connecting to, enabling virtual hosting over TLS.

2.6.3.13 WOLFSSL_ALWAYS_KEEP_SNI Keeps the SNI value in the SSL session after the handshake completes. By default, the SNI data is freed after the handshake to save memory.

2.6.3.14 WOLFSSL_ALWAYS_VERIFY_CB Always invokes the certificate verify callback, even when verification succeeds. By default the callback is only called on failure.

2.6.3.15 HAVE_TRUNCATED_HMAC Enables the Truncated HMAC TLS extension (RFC 6066). Allows using 80-bit HMAC tags instead of the full size to reduce bandwidth.

2.6.3.16 HAVE_TIME_T_TYPE Indicates the platform provides a `time_t` type definition. Set automatically on most platforms.

2.6.3.17 HAVE_SECURE_RENEGOTIATION Enables secure renegotiation support (RFC 5746). Allows TLS connections to renegotiate cipher suites and keys during an active session.

2.6.3.18 HAVE_SERVER_RENEGOTIATION_INFO Enables the server-side renegotiation info extension. Indicates secure renegotiation support in server hello messages.

2.6.3.19 HAVE_SESSION_TICKET Enables TLS session ticket support (RFC 5077). Allows the server to issue session tickets for faster resumption without server-side session state. Required for TLS 1.3 resumption.

2.6.3.20 HAVE_TRUSTED_CA Enables the Trusted CA Indication TLS extension (RFC 6066). Allows the client to indicate which CA certificates it trusts, helping the server select the appropriate certificate chain.

2.6.3.21 HAVE_RPK Enables Raw Public Key support (RFC 7250). Allows using raw public keys instead of X.509 certificates in TLS, reducing handshake overhead for constrained environments.

2.6.3.22 HAVE_ECH Enables Encrypted Client Hello (ECH) support. Encrypts the ClientHello to protect sensitive fields like SNI from passive observers.

2.6.3.23 WOLFSSL_NO_CA_NAMES Disables sending CA names in the CertificateRequest message. Reduces handshake message size when the server has many trusted CAs.

2.6.3.24 WOLFSSL_NO_SERVER_GROUPS_EXT Prevents the server from sending its supported groups in a TLS extension when the server's top preference is not in the client's list.

2.6.3.25 HAVE_FFDHE Enables Finite Field Diffie-Hellman Ephemeral (FFDHE) key exchange using standardized groups from RFC 7919.

2.6.3.26 HAVE_SECRET_CALLBACK Enables the TLS secret callback, allowing applications to receive TLS key material during the handshake. Used for key logging, debugging, and integration with external tools.

2.6.3.27 HAVE_PK_CALLBACKS Enables public key operation callbacks, allowing applications to override the default RSA, ECC, and DH operations with custom implementations (e.g., HSM or secure element integration).

2.6.3.28 WOLFSSL_SNIFFER Enables TLS packet sniffing support. Allows decrypting and inspecting TLS traffic using the wolfSSL sniffer library with the private key.

2.6.3.29 HAVE_WEBSERVER Enables web server-oriented features in wolfSSL, such as additional HTTP helper functions.

2.6.3.30 HAVE_WOLF_EVENT Enables wolf event-driven processing support for async operations. Provides an event queue for managing pending async crypto operations.

2.6.3.31 HAVE_WRITE_DUP Enables write duplication support, allowing separate threads to perform SSL read and write operations simultaneously on the same SSL object.

2.6.3.32 NO_CERTS Disables all certificate processing in wolfSSL. Use for PSK-only configurations where no certificate handling is needed, significantly reducing code size.

2.6.3.33 NO_CLIENT_CACHE Disables the client-side session cache. Only the server session cache will be used for session resumption.

2.6.3.34 WOLFSSL_HAVE_PRF Enables access to the TLS Pseudo-Random Function (PRF). Allows applications to derive additional keying material using the TLS PRF.

2.6.3.35 WOLFSSL_REQUIRE_TCA Requires that the client send the Trusted CA extension. If the extension is missing, the handshake will fail.

2.6.3.36 WOLFSSL_DH_EXTRA Stores additional DH key information in the SSL object. Provides access to DH parameters and keys after the handshake.

2.6.3.37 WOLFSSL_CURVE25519_BLINDING Enables blinding for Curve25519 operations during TLS key exchange. Protects against timing side-channel attacks.

2.6.3.38 WOLFSSL_KEY_GEN Turns on wolfSSL's RSA key generation functionality. See Keys and Certificates for more information.

2.6.3.39 WOLF_PRIVATE_KEY_ID This is used with PKCS11 to enable support for key ID and label API's. FIPS v5 and older doesn't support `WOLF_PRIVATE_KEY_ID` with Crypto Callbacks.

2.6.3.40 WOLFSSL_WOLFSENTRY_HOOKS This switch adds support in the TLS layer for generic network `accept` and `connect` filter hooks, using `wolfSSL_CTX_set_AcceptFilter()` and `wolfSSL_CTX_set_ConnectFilter()`. It also activates wolfSentry integration in the example client and server applications.

2.6.3.41 WOLFSSL_CERT_EXT Certificate extension, key and cert generation feature.

2.6.3.42 WOLFSSL_CERT_REQ Certificate request, key, and cert generation feature.

2.6.3.43 WOLFSSL_SSLKEYLOGFILE This enables the key logging used by Wireshark. It does produce a compiler warning since the master secret and client random are written to a file. This is useful for testing and not recommended for production.

2.6.3.44 WOLFSSL_SSLKEYLOGFILE_OUTPUT This macro defines the filename for the key logging. It is used with `WOLFSSL_SSLKEYLOGFILE`.

2.6.3.45 WOLFSSL_HAVE_WOLFSCEP Enable feature used by autoconf to see if wolfSCEP is available.

2.6.3.46 WOLFSSL_HAVE_MIN This macro is for portability of the library to indicate if MIN/MAX are already defined by the platform. It prevents duplicate definitions.

2.6.3.47 WOLFSSL_HAVE_TLS_UNIQUE Keeps the "Finished" messages after a TLS handshake for use as the "tls-unique" channel binding. Added in libest port: allow applications to get the 'tls-unique' Channel Binding Type (https://tools.ietf.org/html/rfc5929#section-3). This is used in the EST protocol to bind an enrollment to a TLS session through 'proof-of-possession' (https://tools.ietf.org/html/rfc7030#section-3.4 and https://tools.ietf.org/html/rfc7030#section-3.5).

2.6.3.48 WOLFSSL_ENCRYPTED_KEYS Enable for encrypted keys PKCS8 support. This macro enables PKCS8 password based key encryption. Here is a link to RFC PKCS8 documentation (https://datatracker.ietf.org/doc/html/rfc5208).

2.6.3.49 WOLFSSL_CUSTOM_OID Certificate feature that enables custom OID support for subject and request extensions. This also applies to parsing certificates with custom OID.

2.6.3.50 WOLFSSL_RIPEMD Enables RIPEMD-160 support.

2.6.3.51 WOLFSSL_SHA384 Enables SHA-384 support.

2.6.3.52 WOLFSSL_SHA512 Enables SHA-512 support.

2.6.3.53 WOLFSSL_AES_DIRECT Enables direct AES ECB mode support. On its own ECB mode is not considered secure. This feature is required for PKCS7. Warning: In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB API's directly whenever possible.

2.6.3.54 DEBUG_WOLFSSL Builds in the ability to debug. For more information regarding debugging wolfSSL, see Debugging.

2.6.3.55 HAVE_AESCCM Enables AES-CCM support.

2.6.3.56 HAVE_AESGCM Enables AES-GCM support.

2.6.3.57 WOLFSSL_AES_XTS Enables AES-XTS support.

2.6.3.58 WOLFSSL_AES_128 Enables AES-128 key size support. Enabled by default. Disable to remove AES-128 and reduce code size when only larger key sizes are needed.

2.6.3.59 WOLFSSL_AES_192 Enables AES-192 key size support. Enabled by default. Disable to remove AES-192 and reduce code size.

2.6.3.60 WOLFSSL_AES_256 Enables AES-256 key size support. Enabled by default. Disable to remove AES-256 when not needed.

2.6.3.61 AES_MAX_KEY_SIZE Sets the maximum AES key size in bits. Defaults to 256. Can be set to 128 or 192 to reduce code and memory usage.

2.6.3.62 HAVE_AES_ECB Enables AES-ECB (Electronic Codebook) mode. Off by default. ECB mode encrypts each block independently and is generally not recommended for most applications, but is needed by some protocols.

2.6.3.63 HAVE_AES_DECRYPT Enables AES decryption support. On by default. Can be disabled to save code size on platforms that only need AES encryption (e.g., GCM encryption-only).

2.6.3.64 WOLFSSL_AES_COUNTER Enables AES-CTR (Counter) mode. Turns a block cipher into a stream cipher using an incrementing counter.

2.6.3.65 WOLFSSL_AES_CFB Enables AES-CFB (Cipher Feedback) mode. Includes CFB-1, CFB-8, CFB-64, and CFB-128 sub-modes unless restricted by `WOLFSSL_NO_AES_CFB_1_8`.

2.6.3.66 WOLFSSL_NO_AES_CFB_1_8 Disables AES-CFB-1 and AES-CFB-8 sub-modes when `WOLFSSL_AES_CFB` is enabled. Reduces code size when only CFB-64/128 are needed.

2.6.3.67 WOLFSSL_AES_OFB Enables AES-OFB (Output Feedback) mode. OFB mode turns AES into a stream cipher and does not require padding.

2.6.3.68 WOLFSSL_AES_CTS Enables AES-CTS (Ciphertext Stealing) mode. CTS allows encrypting data that is not a multiple of the block size without padding.

2.6.3.69 WOLFSSL_AES_SIV Enables AES-SIV (Synthetic Initialization Vector) mode (RFC 5297). A nonce-misuse resistant AEAD mode that provides deterministic authenticated encryption.

2.6.3.70 WOLFSSL_AES_EAX Enables AES-EAX AEAD mode. EAX is a two-pass AEAD scheme built from CTR mode and OMAC (CMAC), providing authenticated encryption with associated data.

2.6.3.71 HAVE_AES_KEYWRAP Enables AES Key Wrap support (RFC 3394). Used for wrapping (encrypting) cryptographic keys for secure transport.

2.6.3.72 WOLFSSL_AES_CBC_LENGTH_CHECKS Enables strict validation of input data length for AES-CBC operations. When enabled, CBC encrypt/decrypt will return an error if the input length is not a multiple of the AES block size.

2.6.3.73 HAVE_AESGCM_DECRYPT Enables AES-GCM decryption support. On by default when HAVE_AESGCM is enabled. Can be disabled on constrained devices that only need GCM encryption.

2.6.3.74 WOLFSSL_AESGCM_STREAM Enables streaming AES-GCM API. Allows processing AES-GCM data incrementally rather than all at once, useful for large data or memory-constrained environments.

2.6.3.75 WC_AES_GCM_DEC_AUTH_EARLY Authenticates the GCM tag before performing decryption. Provides fail-fast behavior when the authentication tag does not match, avoiding unnecessary decryption of invalid ciphertext.

2.6.3.76 GCM_TABLE Use a pre-computed 4-bit lookup table for AES-GCM Galois field multiplication. Faster than GCM_SMALL but uses more memory. See also GCM_TABLE_4BIT and GCM_WORD32.

2.6.3.77 GCM_TABLE_4BIT Explicit option for 4-bit GCM lookup table mode. Functions similarly to GCM_TABLE.

2.6.3.78 GCM_WORD32 Use a 32-bit word implementation for AES-GCM Galois field multiplication. An alternative to GCM_SMALL and GCM_TABLE that works well on platforms without 64-bit support.

2.6.3.79 GCM_GMULT_LEN Enables GCM GMULT length optimization for processing multiple blocks of AAD or ciphertext in a single GMULT call.

2.6.3.80 WOLFSSL_AESXTS_STREAM Enables streaming AES-XTS API. Allows processing AES-XTS data incrementally across multiple update calls rather than a single operation.

2.6.3.81 WC_AESXTS_STREAM_NO_REQUEST_ACCOUNTING Disables request accounting in the streaming AES-XTS API. Removes overhead of tracking data unit boundaries when not needed.

2.6.3.82 WC_AES_XTS_SUPPORT_SIMULTANEOUS_ENC_AND_DEC_KEYS Allows an AES-XTS context to hold both encryption and decryption keys simultaneously. By default, XTS contexts support only one direction at a time to save memory.

2.6.3.83 HAVE_CAMELLIA Enables Camellia support.

2.6.3.84 HAVE_CHACHA Enables ChaCha20 support.

2.6.3.85 HAVE_POLY1305 Enables Poly1305 support.

2.6.3.86 POLY130564 Uses the 64-bit Poly1305 implementation for better performance on 64-bit platforms. Automatically selected on platforms with 64-bit support.

2.6.3.87 USE_INTEL_POLY1305_SPEEDUP Enables Intel AVX/AVX2 optimized Poly1305 implementation for maximum throughput on Intel/AMD processors.

2.6.3.88 HAVE_CRL Enables Certificate Revocation List (CRL) support.

2.6.3.89 HAVE_CRL_IO Enables blocking inline HTTP request on the CRL URL. It will load the CRL into the `WOLFSSL_CTX` and apply it to all WOLFSSL objects created from it.

2.6.3.90 HAVE_ECC Enables Elliptical Curve Cryptography (ECC) support.

2.6.3.91 HAVE_LIBZ Is an extension that can allow for compression of data over the connection. It normally shouldn't be used, see the note below under configure notes libz.

2.6.3.92 OPENSSL_EXTRA Builds even more OpenSSL compatibility into the library, and enables the wolfSSL OpenSSL compatibility layer to ease porting wolfSSL into existing applications which had been designed to work with OpenSSL. It is off by default.

2.6.3.93 HAVE_EXT_CACHE Enables a feature support use of an external session cache (vs an internal one).

2.6.3.94 WOLFSSL_WPAS_SMALL Enables a smaller subset of the compatibility layer for WPA supplicant support.

2.6.3.95 OPENSSL_ALL Enables support for all compatibility functions for testing integration.

2.6.3.96 OPENSSL_COEXIST OpenSSL compat layer. Needs old names disabled. Mode to allow wolfSSL and OpenSSL to exist together.

2.6.3.97 OPENSSL_VERSION_NUMBER Specifies the version number to implement OpenSSL compatibility.

2.6.3.98 OLD_HELLO_ALLOWED Allows SSLv2-format ClientHello messages for backward compatibility with legacy clients that use the SSLv2 record format to negotiate higher protocol versions.

2.6.3.99 WOLFSSL_NGINX OpenSSL compatibility application specific. Use, nginx `(--enable-nginx) WOLFSSL_NGINX`.

2.6.3.100 WOLFSSL_ERROR_CODE_OPENSSL OpenSSL compatibility API wolfSSL_EVP_PKEY_cmp returns 0 on success and -1 on failure. This behavior is different from OpenSSL. EVP_PKEY_cmp returns: 1: two keys match 0: do not match -1: key types are different -2: the operation is not supported If you want this function to behave the same as openSSL, define WOLFSSL_ERROR_CODE_OPENSSL so that WS_RETURN_CODE translates return codes to match OpenSSL equivalent behavior.

2.6.3.101 WOLFSSL_HARDEN_TLS Implement the recommendations specified in RFC9325. This macro needs to be defined to the desired number of bits of security. The currently implemented values are 112 and 128 bits. The following macros disable certain checks. - WOLFSSL_HARDEN_TLS_ALLOW_TRUNCATED_HMAC - WOLFSSL_HARDEN_TLS_ALLOW_OLD_TLS - WOLFSSL_HARDEN_TLS_NO_SCR_CHECK - WOLFSSL_HARDEN_TLS_NO_PKEY_CHECK - WOLFSSL_HARDEN_TLS_ALLOW_ALL_CIPHERSUITES

2.6.3.102 WOLFSSL_ASIO OpenSSL compatibility specific macro.

2.6.3.103 WOLFSSL_QT OpenSSL compatibility specific. Enable DH Extra for QT, OpenSSL all, OpenSSH, and static ephemeral.

2.6.3.104 WOLFSSL_QNX_CAAM Enables QNX CAAM (Cryptographic Acceleration and Assurance Module) support for hardware-accelerated crypto on QNX-based systems.

2.6.3.105 WOLFSSL_HAPROXY OpenSSL compatibility specific macro.

2.6.3.106 WOLFSSL_ASYNC_IO Used in async cleanup.

2.6.3.107 WOLFSSL_ASYNC_CRYPT_SW Enables software-based async crypto simulation for testing. Simulates async behavior without requiring actual async hardware.

2.6.3.108 WOLFSSL_ATMEL Enables ASF hooks seeding random data using the `atmel_get_random_number` function.

2.6.3.109 WOLFSSL_CMAC Additional CMAC algorithm enable. Note: requires `WOLFSSL_AES_DIRECT`.

2.6.3.110 WOLFSSL_ESPIDF_ERROR_PAUSE Used only in test.c and on test error adds a delay for debugging purposes.

2.6.3.111 TEST_IPV6 Turns on testing of IPv6 in the test applications. wolfSSL proper is IP neutral, but the testing applications use IPv4 by default.

2.6.3.112 TEST_NONBLOCK_CERTS Used only for testing a non-blocking OCSP response. Enabled with `WOLFSSL_NONBLOCK_OCSP` and `OCSP_WANT_READ`.

2.6.3.113 TEST_OPENSSL_COEXIST Use when enabling the build option: `./configure --enable-opensslcoexist`.

2.6.3.114 TEST_PK_PRIVKEY Used for testing PK callbacks only. In wolfssl/test.h it uses the context to pass the actual private key which is loaded and used in the PK callback.

2.6.3.115 TEST_BUFFER_SIZE Allows overriding the TLS benchmarking test buffer size used with the example client/server `-B` option.

2.6.3.116 FORCE_BUFFER_TEST Forces use of the test_certs.h buffers instead of using the file system. Used for internal testing only in wolfssl/test.h.

2.6.3.117 WOLFSSL_FORCE_MALLOC_FAIL_TEST Define for internal testing to induce random malloc failures.

2.6.3.118 WOLFSSL_POST_HANDSHAKE_AUTH TLS extension, Used for post-handshake authentication.

2.6.3.119 WOLFSSL_PSK_MULTI_ID_PER_CS With TLS 1.3 PSK, when `WOLFSSL_PSK_MULTI_ID_PER_CS` is defined, multiple IDs for a cipher suite can be handled.

2.6.3.120 WOLFSSL_PUBLIC_ASN This exposes the ASN.1 API's publicly that are used internally. This is useful for customers who want to use the internal asn.h API's to parse.

2.6.3.121 WOLFSSL_QUIC Enables support for QUIC protocol. See (https://github.com/wolfSSL/wolfssl/blob/maste for more information.

2.6.3.122 WOLFSSL_QUIC_MAX_RECORD_CAPACITY Defines max quic capacity as 1024*1024 – 1 MB.

2.6.3.123 WOLFSSL_RENESAS_FSPSM_TLS Not yet supported TLS related capabilities.

2.6.3.124 WOLFSSL_REFCNT_ERROR_RETURN Returns errors on reference counting failures in SSL objects instead of silently continuing. Helps detect resource management issues.

2.6.3.125 WOLFSSL_RENESAS_TSIP_TLS This is for disabling only the TSIP TLS-linked common key encryption method.

2.6.3.126 WOLFSSL_SM2 Define to use SM ciphers.

2.6.3.127 WOLFSSL_SM3 Define to use SM ciphers.

2.6.3.128 WOLFSSL_SM4 Define to use SM ciphers.

2.6.3.129 WOLFSSL_SM4_CBC SM setting for SM4 CBC.

2.6.3.130 WOLFSSL_SM4_CCM SM settings for SM4 CCM.

2.6.3.131 WOLFSSL_SM4_GCM SM settings for SM4 GCM.

2.6.3.132 WOLFSSL_SNIFFER_CHAIN_INPUT The Chain Input option allows the sniffer to receive its input as a struct iovec list. Rather than a pointer to a raw packet.

2.6.3.133 XSLEEP_MS Used for testing only. It allows defining a custom delay.

2.6.3.134 XSNPRINTF Allows overriding the snprintf function.

2.6.3.135 DEFAULT_TIMEOUT_SEC Used with `HAVE_IO_TIMEOUT` to specify the wolfio.c socket timeout in seconds. This is used by the internal socket code for OCSP and CRL HTTP.

2.6.3.136 HAVE_IO_TIMEOUT Certificate revocation. IO options enable support for connect timeout, but the default is off.

2.6.3.137 HAVE_OCSP Enables Online Certificate Status Protocol (OCSP) support.

2.6.3.138 HAVE_CSHARP Turns on configuration options needed for C# wrapper.

2.6.3.139 HAVE_CURVE25519 Turns on the use of curve25519 algorithm.

2.6.3.140 HAVE_ED25519 Turns on use of the ed25519 algorithm.

2.6.3.141 WOLFSSL_DH_CONST Turns off use of floating point values when performing Diffie Hellman operations and uses tables for `XPOW()` and `XLOG()`. Removes dependency on external math library.

2.6.3.142 WOLFSSL_TRUST_PEER_CERT Turns on the use of trusted peer certificates. This allows for loading in a peer certificate to match with a connection rather than using a CA. When turned on if a trusted peer certificate is matched than the peer cert chain is not loaded and the peer is considered verified. Using CAs is preferred.

2.6.3.143 WOLFSSL_STATIC_MEMORY Turns on the use of static memory buffers and functions. This allows for using static memory instead of dynamic.

2.6.3.144 WOLFSSL_STATIC_MEMORY_LEAN It requires `WOLFSSL_STATIC_MEMORY` to be defined. It uses smaller type sizes for structs requiring memory pool sizes of less than 65k and limits features available, like IO buffers, to reduce footprint size.

2.6.3.145 WOLFSSL_SESSION_EXPORT Turns on the use of DTLS session export and import. This allows for serializing and sending/receiving the current state of a DTLS session.

2.6.3.146 WOLFSSL_SESSION_EXPORT_DEBUG Enables debug logging for session export and import operations. Helps diagnose issues with session serialization.

2.6.3.147 WOLFSSL_SESSION_EXPORT_NOPEER Exports sessions without including peer certificate information. Reduces exported session size when peer cert is not needed.

2.6.3.148 WOLFSSL_ARMASM Turns on the use of ARMv8 hardware acceleration.

2.6.3.149 WC_RSA_NONBLOCK Turns on fast math RSA non-blocking support for splitting RSA operations into smaller chunks of work. Feature is enabled by calling `wc_RsaSetNonBlock()` and checking for `FP_WOULDBLOCK` return code.

2.6.3.150 WC_RSA_BLINDING Used to enable timing resistance.

2.6.3.151 WC_RSA_PSS Enables RSA PSS padding. The only TLS 1.3 RSA padding scheme supported is PSS (per specification). PSS padding uses random padding.

2.6.3.152 WOLFSSL_RSA_VERIFY_ONLY Turns on small build for RSA verify only use. Should be used with the macros WOLFSSL_RSA_PUBLIC_ONLY, WOLFSSL_RSA_VERIFY_INLINE, NO_SIG_WRAPPER, and WOLFCRYPT_ONLY.

2.6.3.153 WOLFSSL_RSA_PUBLIC_ONLY Turns on small build for RSA public key only use. Should be used with the macro WOLFCRYPT_ONLY.

2.6.3.154 WOLFSSL_SHA3 Turns on build for SHA3 use. This is support for SHA3 Keccak for the sizes SHA3-224, SHA3-256, SHA3-384 and SHA3-512. In addition WOLFSSL_SHA3_SMALL can be used to trade off performance for resource use.

2.6.3.155 USE_ECDSA_KEYSZ_HASH_ALGO Will choose a hash algorithm that matches the ephemeral ECDHE key size or the next highest available. This workaround resolves issues with some peers that do not properly support scenarios such as a P-256 key hashed with SHA512.

2.6.3.156 WOLFSSL_ALT_CERT_CHAINS Allows CA's to be presented by peer, but not part of a valid chain. Default wolfSSL behavior is to require validation of all presented peer certificates. This also allows loading intermediate CA's as trusted and ignoring no signer failures for CA's up the chain to root. The alternate certificate chain mode only requires that the peer certificate validate to a trusted CA.

2.6.3.157 WOLFSSL_SYS_CA_CERTS Allows wolfSSL to use trusted system CA certificates for verification when wolfSSL_CTX_load_system_CA_certs() is called, either by loading them into wolfSSL certificate manager, or by invoking system authentication APIs. See wolfSSL_CTX_load_system_CA_certs() for more details. This preprocessor macro is automatically set by the --enable-sys-ca-certs configure option.

2.6.3.158 WOLFSSL_SYS_CRYPTO_POLICY Honors system-level crypto policy settings (e.g., /etc/crypto-policies) for restricting available algorithms and key sizes.

2.6.3.159 WOLFSSL_APPLE_NATIVE_CERT_VERIFICATION Enables the use of Apple's native trust APIs when authenticating TLS peer certificates. Requires WOLFSSL_SYS_CA_CERTS to be defined. This macro does not need to be set by the user if building with configure or CMake on iOS or other apple devices, but should be explicitly set on MacOS if you wish to use the native verification methods.

2.6.3.160 WOLFSSL_TEST_APPLE_NATIVE_CERT_VALIDATION Enables testing mode for Apple native certificate validation. Used for unit testing the Apple cert validation integration.

2.6.3.161 WOLFSSL_CUSTOM_CURVES Allow non-standard curves. Includes the curve "a" variable in calculation. Additional curve types can be enabled using HAVE_ECC_SECPR2, HAVE_ECC_SECPR3, HAVE_ECC_BRAINPOOL and HAVE_ECC_KOBLITZ.

2.6.3.162 HAVE_COMP_KEY Enables ECC compressed key support.

2.6.3.163 WOLFSSL_EXTRA_ALERTS Enables additional alerts to be sent during a TLS connection. This feature is also enabled automatically when --enable-opensslextra is used.

2.6.3.164 WOLFSSL_EXTRA Enables extra SSL session information tracking and APIs beyond the standard set. Provides additional session details for debugging and monitoring.

2.6.3.165 WOLFSSL_DEBUG_TLS Enables additional debugging print outs during a TLS connection

2.6.3.166 WOLFSSL_DEBUG_TRACE_ERROR_CODES Enables tracing of error code origins for debugging. Logs where error codes are generated in the wolfSSL source code.

2.6.3.167 WOLFSSL_DEBUG_MEMORY Enables memory allocation debugging. Logs `malloc()` and `free()` calls with file name and line number information.

2.6.3.168 WOLFSSL_DEBUG_OPENSSL Enables debug logging for the OpenSSL compatibility layer functions. Helps trace which OpenSSL compatibility APIs are being called.

2.6.3.169 HAVE_BLAKE2 Enables Blake2b algorithm support

2.6.3.170 HAVE_FALLBACK_SCSV Enables Signaling Cipher Suite Value(SCSV) support on the server side. This handles the cipher suite 0x56 0x00 sent from a client to signal that no downgrade of TLS version should be allowed.

2.6.3.171 HAVE_AEAD Implements the use of AEAD and is required for TLS 1.3.

2.6.3.172 HAVE_AES_CBC Enable option for AES CBC.

2.6.3.173 HAVE_ALPN Crypto enables the option for application-layer protocol negotiation.

2.6.3.174 HAVE_CAVIUM_OCTEON_SYNC This enables the blocking (synchronous) version of the Marvell Cavium/Octeon hardware.

2.6.3.175 HAVE_CERTIFICATE_STATUS_REQUEST Used for Certificate revocation as a cert status request feature.

2.6.3.176 HAVE_CERTIFICATE_STATUS_REQUEST_V2 Used for Certificate revocation as a cert status request feature.

2.6.3.177 HAVE_CURL Used for building a subset of the wolfSSL library when linking with cURL.

2.6.3.178 HAVE_CURVE448 Define for Curve448 support. Additional macro settings can be changed. The default is to enable shared secret, key export, and import.

2.6.3.179 HAVE_CURVE448_SHARED_SECRET Enables Curve448 shared secret computation. On by default when HAVE_CURVE448 is enabled.

2.6.3.180 HAVE_CURVE448_KEY_EXPORT Enables Curve448 public key export. On by default when HAVE_CURVE448 is enabled.

2.6.3.181 HAVE_CURVE448_KEY_IMPORT Enables Curve448 public key import. On by default when HAVE_CURVE448 is enabled.

2.6.3.182 WOLFSSL_ECDHX_SHARED_NOT_ZERO Validates that ECDH shared secrets are not all zeros. Provides protection against invalid curve attacks.

2.6.3.183 HAVE_DANE This option is only supported with HAVE_RPK (Raw Public Keys) and is a placeholder for when it might be added in the future.

2.6.3.184 HAVE_DILITHIUM Enable to include DILITHIUM post quantum cryptography/signature algo.

2.6.3.185 HAVE_DH_DEFAULT_PARAMS Includes default DH parameters for key exchange when the application does not explicitly load its own DH parameters.

2.6.3.186 HAVE_ED25519_KEY_IMPORT ED25519 config. Enables Ed255519 and Curve25519 options for granular control of sign, verify, shared secret, import, and export.

2.6.3.187 HAVE_EX_DATA Enable "extra" EX data APIs for user information in CTX/WOLFSSL.

2.6.3.188 HAVE_EX_DATA_CLEANUP_HOOKS Set the extra data and cleanup callback against the RSA key at an index.

2.6.3.189 HAVE_EX_DATA_CRYPTO Enables extra data (ex_data) support for wolfCrypt objects such as RSA and ECC keys, in addition to SSL/CTX/X509 objects.

2.6.3.190 HAVE_FALCON Enables post-quantum crypto FALCON from OpenQuantumSafe.

2.6.3.191 HAVE_FIPS Used when implementing different FIPS versions.

2.6.3.192 HAVE_FUZZER Enables fuzzing callback support for security testing. Allows a callback to be set that can modify or inspect TLS records during processing.

2.6.3.193 HAVE_KEYING_MATERIAL Enables exporting keying material based on section 7.5 of RFC 8446.

2.6.3.194 HAVE_OID_DECODING Included in ASN template code. Used to decode in some cases.

2.6.3.195 HAVE_MAX_FRAGMENT Sets maximum fragment size. TLS extension.

2.6.3.196 HAVE_MEMCACHED Enables APIs and behaviors needed for memcached compatibility with wolfSSL.

2.6.3.197 WOLFSSL_PSK_ONE_ID Enables support for only one PSK ID with TLS 1.3.

2.6.3.198 WOLFSSL_PSK_IDENTITY_ALERT Sends a specific TLS alert when PSK identity lookup fails during the handshake, rather than a generic handshake failure.

2.6.3.199 SHA256_MANY_REGISTERS A SHA256 version that keeps all data in registers and partially unrolls loops.

2.6.3.200 WOLFCRYPT_HAVE_SRP Enables wolfCrypt secure remote password support

2.6.3.201 WOLFSSL_MAX_STRENGTH Enables the strongest security features only and disables any weak or deprecated features. This results in slower performance due to near constant-time execution to protect against timing-based side-channel attacks.

2.6.3.202 MAX_RECORD_SIZE Determine maximum record size. 2^14 is the max size by standard.

2.6.3.203 MAX_CERTIFICATE_SZ Defines the max size of a certificate message payload assumes MAX_CHAIN_DEPTH number of certificates at 2kb per certificate.

2.6.3.204 MAX_CHAIN_DEPTH Defines the max chain depth.

2.6.3.205 MAX_CIPHER_NAME Defines max cipher name.

2.6.3.206 MAX_DATE_SIZE Defines max size of date either used as byte lastdate, or byte nextdate.

2.6.3.207 MAX_EARLY_DATA_SZ Used to define the maximum early data size.

2.6.3.208 MAX_EX_DATA Sets the maximum number of extra data (ex_data) entries that can be stored per SSL/CTX/X509 object. Default is 5 if not defined.

2.6.3.209 WOLFSSL_MAX_SEND_SZ Define to specify max send size.

2.6.3.210 WOLFSSL_MAX_SUITE_SZ Define to specify max suite size. If too small error out.

2.6.3.211 MAX_WOLFSSL_FILE_SIZE 4 MB allocated size limit.

2.6.3.212 WOLFSSL_MAXQ10XX_TLS Lets maxq10xx know what TLS version we are using.

2.6.3.213 WOLFSSL_MAX_SIGALGO Enables the ability to override maximum signature algorithms.

2.6.3.214 WOLFSSL_MEM_GUARD Can assign a specified memory guard.

2.6.3.215 WOLFSSL_STATIC_EPHEMERAL TLS sniffer support.

2.6.3.216 SSL_SNIFFER_EXPORTS WIN32 sniffer export.

2.6.3.217 WOLFSSL_SNIFFER_KEYLOGFILE The SSL Keylog file option enables the sniffer to decrypt TLS traffic using the master secret obtained from a NSS keylog file. This allows the sniffer to decrypt all TLS traffic, even for TLS connections using ephemeral cipher suites. Keylog file sniffing is supported for TLS versions 1.2 and 1.3. WolfSSL can be configured to export a keylog file using the `--enable-keylog-export` configure option, independently from the sniffer feature (NOTE: never do this in a production environment, as it is inherently insecure). To enable sniffer support for keylog files, use the following configure command line and build as before: ./configure –enable-sniffer CPPFLAGS=-DWOLFSSL_SNIFFER_KEYLOGFILE'.

2.6.3.218 WOLFSSL_SNIFFER_STORE_DATA_CB The Store Data Callback option allows the sniffer to take a callback that is called when storing the application data into a custom buffer rather than into the reallocated data pointer. The callback is called in a loop until all data is consumed. To enable this option, use the following configure command line and build as before: `./configure --enable-sniffer CPPFLAGS=-DWOLFSSL_SNIFFER_STORE_DATA_CB`.

2.6.3.219 WOLFSSL_SNIFFER_WATCH The Session Watching option allows the sniffer to watch any packet provided without initial setup. It will start to decode all TLS sessions and when the server's certificate is detected, the certificate is given to a callback function provided by the user which should provide the appropriate private key. To enable this option, use the following configure command line and build as before: `./configure --enable-sniffer CPPFLAGS=-DWOLFSSL_SNIFFER_WATCH`.

2.6.3.220 STATIC_BUFFER_LEN Don't fragment memory from the record header. Expands to: `RECORD_HEADER_SZ`.

2.6.3.221 STATIC_CHUNKS_ONLY The user has the option to turn off the 16K output option if we are using small static buffers (the default) and SSL_write tries to write data larger than the record we have, dynamically getting it unless the user says only write in static buffer chunks.

2.6.3.222 WOLFSSL_DEF_PSK_CIPHER Enables user-defined PSK cipher.

2.6.3.223 WOLFSSL_OLD_PRIME_CHECK Enable feature which uses faster DH and RSA prime checking.

2.6.3.224 WOLFSSL_OLD_SET_CURVES_LIST Uses the old-style curve list parsing for backward compatibility with applications that set curves using the older format.

2.6.3.225 WOLFSSL_OLD_TIMINGPADVERIFY Uses the old timing-based padding verification for CBC cipher suites. The new method provides better constant-time behavior.

2.6.3.226 WOLFSSL_OLDTLS_AEAD_CIPHERSUITES Enables AEAD cipher suites (GCM, CCM) for TLS versions prior to 1.2. These suites are normally only available in TLS 1.2 and later.

2.6.3.227 WOLFSSL_OLDTLS_SHA2_CIPHERSUITES Enables SHA-2 based cipher suites for TLS versions prior to 1.2. These suites are normally only available in TLS 1.2 and later.

2.6.3.228 WOLFSSL_STATIC_RSA Static ciphers are strongly discouraged and should never be used if avoidable. However there are still legacy systems that ONLY support static cipher suites. To that end if you need to connect to a legacy peer only supporting static RSA cipher suites use this to enable support for static RSA in wolfSSL. (See also `WOLFSSL_STATIC_PSK` and `WOLFSSL_STATIC_DH`)

2.6.3.229 WOLFSSL_STRONGEST_HASH_SIG Prefers the strongest available hash algorithm when performing signature operations during the TLS handshake.

2.6.3.230 WOLFSSL_STATIC_PSK Feature that enables static PSK cipher suites. Static ciphers are highly discouraged see `WOLFSSL_STATIC_RSA`

2.6.3.231 WOLFSSL_STATIC_DH Feature that enables static ECDH cipher suites. Static ciphers are highly discouraged see `WOLFSSL_STATIC_RSA`

2.6.3.232 HAVE_NULL_CIPHER Turns on support for NULL ciphers. This option is highly discouraged from a security standpoint however some systems are too small to perform encrypt/decrypt operations and it is better to at least authenticate messages and peers to prevent message tampering than nothing at all!

2.6.3.233 HAVE_ANON Turns on support for anonymous cipher suites. (Never recommended, some valid use cases involving closed or private networks detached from the web)

2.6.3.234 HAVE_ATEXIT Registers `wolfSSL_Cleanup()` as an `atexit()` handler for automatic cleanup when the program exits.

2.6.3.235 HAVE_LIBOQS Turn on support for the OpenQuantumSafe team's liboqs integration. Please see the appendix "Experimenting with Quantum-Safe Cryptography" in this document for more details.

2.6.3.236 HAVE_LIGHTY Enables APIs and behaviors needed for lighttpd web server compatibility with wolfSSL.

2.6.3.237 WOLFSSL_SP_4096 Enable RSA/DH 4096-bit Single-Precision (SP) support.

2.6.3.238 WOLFSSL_SP_384 Enable ECC SECP384R1 Single-Precision (SP) support. Only applies to `WOLFSSL_SP_MATH`.

2.6.3.239 WOLFSSL_SP_1024 Enable SAKKE pairing based cryptography Single-Precision (SP) support.

2.6.3.240 ATOMIC_USER Enable Atomic Record Layer callbacks.

2.6.3.241 BIG_ENDIAN_ORDER Endianness - defaults to little endian, i.e., makes big-endian.

2.6.3.242 WOLFSSL_32BIT_MILLI_TIME Function TimeNowInMilliseconds() returns an unsigned 32-bit value. The default behavior is to return a signed 64-bit value.

2.6.3.243 WOLFSSL_MAX_DHKEY_BITS DH maximum bit size must be a multiple of 8. DH maximum bit size must not exceed 16384 or greater than `WOLFSSL_MIN_DHKEY_BITS`.

2.6.3.244 WOLFSSL_MIN_DHKEY_BITS The DH minimum bit size must be a multiple of 8. For 112 bits of security, DH needs at least 2048-bit keys, and the minimum bit size must not be greater than 16000.

2.6.3.245 WOLFSSL_MAX_MTU Max expected MTU. 1500 - 100 bytes to account for UDP and IP headers.

2.6.3.246 IGNORE_NETSCAPE_CERT_TYPE Define to input netscape cert type but holds a place.

2.6.3.247 SESSION_CERTS TLS session cache for certs.

2.6.3.248 WOLFSSL_DUAL_ALG_CERTS Dual algorithm certificate required feature.

2.6.3.249 CRL_MAX_REVOKED_CERTS Specifies the number of buffers to hold RevokedCerts. The default value is set to 4.

2.6.3.250 CRL_STATIC_REVOKED_LIST Enables a fixed static list of RevokedCerts to allow for a binary search.

2.6.3.251 SESSION_INDEX Identifies the session's location in the cache. Specifies index session/row shifts.

2.6.3.252 SESSION_TICKET_HINT_DEFAULT The ticket hint default is used to set the default hint value. The Ticket Key lifetime must be longer than the ticket life hint.

2.6.3.253 WOLFSSL_DTLS13 Enable wolfSSL DTLS 1.3.

2.6.3.254 WOLFSSL_TLS13 Enable TLS 1.3 protocol implementation.

2.6.3.255 WOLFSSL_TLS13_IGNORE_AEAD_LIMITS Limits specified by https://www.rfc-editor.org/rfc/rfc9147.html#
aead-limits. We specify the limit by which we need to do a key update as the halfway point to the hard decryption fail limit.

2.6.3.256 WOLFSSL_TLS13_DRAFT Uses draft TLS 1.3 specification parameters for testing against draft implementations. Not for production use.

2.6.3.257 WOLFSSL_TLS13_MIDDLEBOX_COMPAT Enable middlebox compatibility in the TLS 1.3 handshake. This includes sending ChangeCipherSpec before encrypted messages and including a session ID.

2.6.3.258 WOLFSSL_TLS13_IGNORE_PT_ALERT_ON_ENC Ignores plaintext alerts received when encrypted records are expected in TLS 1.3. May improve interoperability with some implementations.

2.6.3.259 WOLFSSL_TLS13_SHA512 Allow generation of SHA-512 digests in handshake - no ciphersuite requires SHA-512 at this time. This enables calculation of a SHA2-512 hash for the handshake messages even though its not used by TLS v1.3 yet.

2.6.3.260 WOLFSSL_TLS13_TICKET_BEFORE_FINISHED Allows the server to send a NewSessionTicket message before receiving the client's Finished message. See TLS 1.3 specification, Section 4.6.1, Paragraph 4.

2.6.3.261 WOLFSSL_EARLY_DATA Enables TLS 1.3 0-RTT (Zero Round Trip Time) early data support. Allows clients to send application data in the first flight of the handshake for faster connection establishment. Requires session resumption via PSK or session tickets.

2.6.3.262 WOLFSSL_EARLY_DATA_GROUP Groups the early data message with the ClientHello when sending. Reduces the number of network round trips by combining messages.

2.6.3.263 WOLFSSL_CHECK_SIG_FAULTS Verifies the ECC signature after signing to detect fault injection attacks. Useful in environments where hardware fault attacks are a concern.

2.6.3.264 WOLFSSL_CIPHER_INTERNALNAME Uses wolfSSL internal cipher suite names instead of IANA-standard names when reporting cipher suite information through the API.

2.6.3.265 WOLFSSL_PSK_ID_PROTECTION Enables PSK identity protection in TLS 1.3. Encrypts the PSK identity to prevent passive observers from tracking clients by their PSK identity.

2.6.3.266 WOLFSSL_NO_CLIENT_CERT_ERROR When enabled, the server requires the client to send a valid certificate. If the client does not provide one, the handshake fails with an error.

2.6.3.267 WOLFSSL_NONBLOCK_OCSP Enables non-blocking OCSP stapling processing. Allows OCSP lookups to be performed asynchronously during the TLS handshake.

2.6.3.268 WOLFSSL_TLS_OCSP_MULTI Enables support for multiple OCSP responses in TLS, allowing stapling of OCSP responses for intermediate certificates in addition to the end-entity certificate.

2.6.3.269 WOLFSSL_CERT_SETUP_CB Enables a certificate setup callback that is invoked during the TLS 1.3 handshake. Allows dynamic certificate and key selection based on the ClientHello contents.

2.6.3.270 WOLFSSL_RW_THREADED Enables read/write threading support, allowing separate threads to perform TLS read and write operations concurrently on the same SSL session.

2.6.3.271 WOLFSSL_PRIORITIZE_PSK During a TLS 1.3 handshake, prioritizes PSK order instead of ciphersuite order when selecting a cipher suite. The PSK callback order determines preference.

2.6.3.272 WOLFSSL_UIP When CONTIKI is defined, it is an implementation of UIP.

2.6.3.273 TLS13_MAX_TICKET_AGE Specifies Max ticket age. For TLS 1.3, this is 7 days.

2.6.3.274 TLS13_TICKET_NONCE_STATIC_SZ TLS13_TICKET_NONCE_STATIC_SZ is not supported in this `FIPS_VERSION_GE`.

2.6.3.275 TIME_OVERRIDES Application provides custom time functions (`XTIME`, `XGMTIME`, etc.) instead of using the system time functions.

2.6.3.276 TLS13_TICKET_NONCE_MAX_SZ Defines version max size for ticket nonce. Max size is defined as 255 bytes.

2.6.3.277 WOLFSSL_TICKET_ENC_AES128_GCM Use AES128-GCM to encrypt/decrypt session tickets in the default callback. This is server-only. If ChaCha20/Poly1305 is not compiled, this is the default algorithm.

2.6.3.278 WOLFSSL_TICKET_ENC_AES256_GCM Use AES256-GCM to encrypt/decrypt session tickets in default callback. Server only.

2.6.3.279 WOLFSSL_TICKET_ENC_CHACHA20_POLY1305 Use ChaCha20-Poly1305 to encrypt/decrypt session tickets in the default callback. If none are defined, the default algorithm is used, and algorithms are compiled. This is server-only.

2.6.3.280 WOLFSSL_TICKET_ENC_CBC_HMAC Uses CBC+HMAC for session ticket encryption instead of an AEAD cipher. Provides an alternative for builds without AEAD support.

2.6.3.281 WOLFSSL_TICKET_EXTRA_PADDING_SZ Defines ticket extra padding size defined as 32.

2.6.3.282 WOLFSSL_TICKET_HAVE_ID Use to make sure the ticket has ID. Only add to the cache when support is built in and when the ticket contains an ID. Otherwise we have no way to actually retrieve a ticket from the cache.

2.6.3.283 WOLFSSL_TICKET_KEY_LIFETIME The default lifetime is 1 hour from the issue of the first ticket with the key. It must be longer than a hint.

2.6.3.284 WOLFSSL_TICKET_NONCE_MALLOC Enable dynamic allocation of ticket nonces. Need to disable the HKDF expand callbacks.

2.6.3.285 SHOW_CERTS Show certs will output certs when defined. Use for embedded debugging.

2.6.3.286 SHOW_SECRETS Used for debugging. It will show applicable secrets.

2.6.3.287 SHOW_SIZES Displays sizes of major wolfSSL structures at initialization. Useful for debugging and memory analysis on constrained systems.

2.6.3.288 DEBUG_UNIT_TEST_CERTS Used when debugging name constraint tests. Not static to allow use in multiple locations with complex define guards.

2.6.3.289 DEBUG_WOLFSSL_VERBOSE When using the `OPENSSL_EXTRA` or `DEBUG_WOLFSSL_VERBOSE` macro, `WOLFSSL_ERROR` is mapped to the new function `WOLFSSL_ERROR_LINE`, which gets the line number and function name where `WOLFSSL_ERROR` is called.

2.6.3.290 SOCKET_INVALID Used to define an invalid socket and is defined as -1.

2.6.3.291 WOLFSSL_SOCKET_INVALID Used for testing and it only allows overriding the value used to indicate an invalid socket. Typically is -1.

2.6.3.292 WOLFSSL_SOCKET_IS_INVALID Used in socket handling.

2.6.3.293 WOLFSSL_SRTP Used to activate SRTP.

2.6.3.294 WOLFSSL_CIPHER_CHECK_SZ Defined as cipher check size which needs 64-bits to confirm encrypt operation worked.

2.6.3.295 DTLS_CID_MAX_SIZE DTLS 1.3 parsing code copies the record header in a static buffer to decrypt the record. Increasing the CID max size also increases this buffer, impacting the per-session runtime memory footprint. The max size for DTLS CID is 255 bytes.

2.6.3.296 DTLS13_EPOCH_SIZE Portability improvement with DTLS 1.3 epoch. Implements a way to save the key bound to a DTLS epoch and setting the right key/epoch when needed.

2.6.3.297 DTLS13_RETRANS_RN_SIZE Portability improvement with DTLS 1.3. Used in DTLS 1.3 to identify size before retransmission.

2.6.3.298 WOLFSSL_DTLS_FRAG_POOL_SZ Defines the allowed number of fragments per specified time.

2.6.3.299 WOLFSSL_DTLS_MTU Enables DTLS MTU (Maximum Transmission Unit) management APIs for controlling the maximum datagram size during DTLS communication.

2.6.3.300 WOLFSSL_CLIENT_SESSION_DEFINED Declare opaque struct for API to use.

2.6.3.301 WOLFSSL_COND Defined if this system supports signaling COND_TYPE - type that should be passed into the signaling API.

2.6.3.302 WOLFSSL_DTLS_CH_FRAG Allows a server to process a fragmented second/verified (one containing a valid cookie response) ClientHello message. The first/unverified (one without a cookie extension) ClientHello MUST be unfragmented so that the DTLS server can process it statelessly. This is only implemented for DTLS 1.3. The user MUST call `wolfSSL_dtls13_allow_ch_frag()` on the server to explicitly enable this during runtime. Note: Using DTLS 1.3 + pqc without `WOLFSSL_DTLS_CH_FRAG` will probably fail In this case use `--enable-dtls-frag-ch` to enable it.

2.6.3.303 WOLFSSL_DTLS_MTU_ADDITIONAL_READ_BUFFER We need additional bytes to read so that we can work with a peer who has a slightly different MTU than us.

2.6.3.304 WOLFSSL_DTLS_WINDOW_WORDS Used to check storage size or to verify if an index is valid for window.

2.6.3.305 WOLFSSL_EXPORT_SPC_SZ Define to specify the amount of bytes used from CipherSpecs.

2.6.3.306 WOLFSSL_MIN_DOWNGRADE Specifies minimum downgrade version.

2.6.3.307 WOLFSSL_MIN_DTLS_DOWNGRADE Specifies minimum DTLS downgrade version.

2.6.3.308 WOLFSSL_MIN_ECC_BITS Can set minimum ECC key size allowed.

2.6.3.309 WOLFSSL_MIN_RSA_BITS By default, wolfSSL restricts RSA key sizes to 1024-bits minimum. To allow the decoding of smaller, less secure RSA keys like 512-bit keys, you will need to add the compiler flag `-DWOLFSSL_MIN_RSA_BITS=512` to CFLAGS or CPPFLAGS or define it in your user_settings header.

2.6.3.310 WOLFSSL_MODE_AUTO_RETRY_ATTEMPTS Used to limit the possibility of an infinite retry loop.

2.6.3.311 WOLFSSL_MULTICAST DTLS multicast feature.

2.6.3.312 WOLFSSL_MULTICAST_PEERS Multicast feature defined as max allowed 100 peers.

2.6.3.313 WOLFSSL_MYSQL_COMPATIBLE Enables MySQL protocol compatibility behaviors in wolfSSL. Required when using wolfSSL as the TLS provider for MySQL.

2.6.3.314 WOLFSSL_NAMES_STATIC Uses static ECC structs for Position Independent Code (PIC).

2.6.3.315 WOLFSSL_SEND_HRR_COOKIE TLS extension used by DTLS 1.3.

2.6.3.316 WOLFSSL_SEP Feature certificate policy set extension.

2.6.3.317 WOLFSSL_SECURE_RENEGOTIATION_ON_BY_DEFAULT Enables secure renegotiation by default for all new SSL contexts without requiring an explicit API call.

2.6.3.318 WOLFSSL_SESSION_ID_CTX Used to copy over application session context ID.

2.6.3.319 WOLFSSL_SESSION_TIMEOUT Default session resumption cache timeout in seconds is used to define timeout manually.

2.6.3.320 WOLFSSL_SET_CIPHER_BYTES Enables setting cipher suites by raw two-byte values instead of name strings. Useful for programmatic cipher suite configuration.

2.6.3.321 KEEP_OUR_CERT Used to ensure the ability to return SSL certificate.

2.6.3.322 KEEP_PEER_CERT Retains peer certs. Parts of the OpenSSL compatibility layer require peer certs.

2.6.3.323 WOLFSSL_SIGNER_DER_CERT This enables retention of the DER/ASN.1 used for signing. This is used by the compatibility layer an example of this is `wolfSSL_X509_STORE_get1_certs`.

2.6.3.324 CA_TABLE_SIZE Used by the wolfSSL Certificate Manager signer table. The default `CA_TABLE_SIZE` is 11, but this can be adjusted based on actual needs. Each `WOLFSSL_CTX` has its own Certificate Manager (CM).

2.6.3.325 ECDHE_SIZE Define to allow this to be overridden at compile-time. ECDHE server size defaults to 256 bits, which can set a predetermined ECDHE curve size. The default is 32 bytes.

2.6.3.326 CIPHER_NONCE It is used as a cryptographic number, which is implemented in authentication. It is a pseudo-random number, which is an integrity-only cipher suite.

2.6.3.327 WOLFSSL_USE_POPEN_HOST Uses popen for creating socket with host and post for wolfio.c socket open code used with CRL and OCSP.

2.6.3.328 CloseSocket A way to override the function used for closing a socket. Used with CRL, OCSP and BIO.

2.6.3.329 CONFIG_POSIX_API Enables POSIX names for networking systems calls.

2.6.3.330 WOLFSSL_USER_CURRTIME Add in the option to use in test.h without the gettimeofday function using the macro `WOLFSSL_USER_CURRTIME`.

2.6.3.331 WOLFSSL_USER_MUTEX Option for user-defined mutexes with `WOLFSSL_USER_MUTEX`.

2.6.3.332 DEFAULT_MIN_ECCKEY_BITS Identifies the minimum number of bits in ECCkey.

2.6.3.333 DEFAULT_MIN_RSAKEY_BITS Identifies the minimum number of bits in RSA key.

2.6.3.334 EXTERNAL_SERIAL_SIZE A raw serial number byte that writes X509 serial numbers in unsigned binary to a buffer. For all cases, the buffer needs to be at least `EXTERNAL_SERIAL_SIZE` (32). On success, it returns `WOLFSSL_SUCCESS`. Note: this is a internal macro that cannot be user defined.

2.6.3.335 LARGE_STATIC_BUFFERS Embedded callbacks require large static buffers; make sure it gives the option to enable larger buffers to 16K.

2.6.3.336 LIBWOLFSSL_VERSION_STRING This is the wolfSSL version string populated for release bundles or when `./configure` is run. There is also a 32-bit HEX version of this in `LIBWOLFSSL_VERSION_HEX`. These come from wolfssl/version.h.

2.6.4 Customizing or Porting wolfSSL

2.6.4.1 WOLFSSL_USER_SETTINGS If defined, it allows the use of a user-specific settings file. The file must be named `user_settings.h` and exist in the include path. It is included prior to the standard `settings.h` file, so default settings can be overridden.

2.6.4.2 WOLFSSL_CALLBACKS This extension allows debugging callbacks through the use of signals in an environment without a debugger. It is off by default. It can also be used to set up a timer with blocking sockets. Please see Callbacks for more information.

2.6.4.3 WOLF_CRYPTO_CB Enable crypto callback support. This feature is also enabled automatically when `--enable-cryptocb` is used.

2.6.4.4 WOLF_CRYPTO_CB_FIND Enable find device callback functions for looking up registered crypto devices by device ID. Requires `WOLF_CRYPTO_CB`.

2.6.4.5 WOLF_CRYPTO_CB_CMD Enable command callback functions that are invoked during register and unregister of crypto callback devices. Requires `WOLF_CRYPTO_CB`.

2.6.4.6 WOLF_CRYPTO_CB_COPY Enable copy callback for algorithm structures, allowing hash and cipher state to be copied via the crypto callback framework. Requires `WOLF_CRYPTO_CB`.

2.6.4.7 WOLF_CRYPTO_CB_FREE Enable free callback for algorithm structures, allowing cleanup of crypto objects via the crypto callback framework. Requires `WOLF_CRYPTO_CB`.

2.6.4.8 WOLF_CRYPTO_CB_AES_SETKEY Enable crypto callback support for AES key setup operations. Allows hardware to handle key scheduling. Requires `WOLF_CRYPTO_CB`.

2.6.4.9 WOLF_CRYPTO_CB_RSA_PAD Enable crypto callback support for RSA padding operations, allowing custom padding handling by hardware or external modules. Requires WOLF_CRYPTO_CB.

2.6.4.10 DEBUG_CRYPTOCB Enable debug InfoString functions for crypto callback operations. Useful for debugging which crypto operations are being routed to hardware.

2.6.4.11 WC_USE_DEVID Specify a default device ID to use for crypto callbacks when no hardware-specific device (such as CAAM) is detected.

2.6.4.12 WC_NO_DEFAULT_DEVID Disable automatic default device ID selection in the crypto callback framework. When defined, applications must explicitly pass a device ID for all crypto operations.

2.6.4.13 WOLFSSL_CAAM_DEVID Defines the device ID constant (value 7) for NXP CAAM hardware crypto. Used in default device ID selection logic.

2.6.4.14 NO_SHA2_CRYPTO_CB Disable crypto callback support for SHA-384 and SHA-512 operations. When defined, these hash operations will always use software implementations.

2.6.4.15 WOLF_CRYPTO_CB_ONLY_RSA Restricts RSA operations to use only crypto callbacks, disabling all software RSA implementations. Useful when RSA should be delegated entirely to hardware.

2.6.4.16 WOLFSSL_DYN_CERT Allow allocation of subjectCN and publicKey fields when parsing certificates even with WOLFSSL_NO_MALLOC set. If using the WOLFSSL_NO_MALLOC option with RSA certificates the public key needs to be retained for CA's for validate certificates on the peer's certificate. This appears as a ConfirmSignature error -173 BAD_FUNC_ARG, since the ca->publicKey is NULL.

2.6.4.17 WOLFSSL_USER_IO Allows the user to remove automatic setting of the default I/O functions EmbedSend() and EmbedReceive(). Used for custom I/O abstraction layer (see Abstraction Layers for more details).

2.6.4.18 NO_FILESYSTEM Is used if stdio isn't available to load certificates and key files. This enables the use of buffer extensions to be used instead of the file ones.

2.6.4.19 NO_INLINE Disables the automatic inlining of small, heavily used functions. Turning this on will slow down wolfSSL and actually make it bigger since these are small functions, usually much smaller than function call setup/return. You'll also need to add wolfcrypt/src/misc.c to the list of compiled files if you're not using autoconf.

2.6.4.20 NO_DEV_RANDOM Disables the use of the default /dev/random random number generator. If defined, the user needs to write an OS-specific GenerateSeed() function (found in wolfcrypt/src/random.c).

2.6.4.21 NO_MAIN_DRIVER Is used in the normal build environment to determine whether a test application is called on its own or through the testsuite driver application. You'll only need to use it with the test files: test.c, client.c, server.c, echoclient.c, echoserver.c, and testsuite.c

2.6.4.22 NO_WRITEV Disables simulation of writev() semantics.

2.6.4.23 SINGLE_THREADED Is a switch that turns off the use of mutexes. wolfSSL currently only uses one for the session cache. If your use of wolfSSL is always single threaded you can turn this on.

2.6.4.24 USER_TICKS Allows the user to define their own clock tick function if time(0) is not wanted. Custom function needs second accuracy, but doesn't have to be correlated to Epoch. See `LowResTimer()` function in `wolfssl_int.c`.

2.6.4.25 USER_TIME Disables the use of time.h structures in the case that the user wants (or needs) to use their own. See `wolfcrypt/src/asn.c` for implementation details. The user will need to define and/or implement `XTIME()`, `XGMTIME()`, and `XVALIDATE_DATE()`.

2.6.4.26 USE_CERT_BUFFERS_256 Enables 256-bit test certificate and key buffers located in `<wolfssl_root>/wolfssl/certs_test.h`. Helpful when testing on and porting to embedded systems with no filesystem.

2.6.4.27 USE_CERT_BUFFERS_1024 Enables 1024-bit test certificate and key buffers located in `<wolfssl_root>/wolfssl/certs_test.h`. Helpful when testing on and porting to embedded systems with no filesystem.

2.6.4.28 USE_CERT_BUFFERS_2048 Enables 2048-bit test certificate and key buffers located in `<wolfssl_root>/wolfssl/certs_test.h`. Helpful when testing on and porting to embedded systems with no filesystem.

2.6.4.29 USE_CERT_BUFFERS_3072 Enables 3072-bit test certificate and key buffers located in `<wolfssl_root>/wolfssl/certs_test.h`. Helpful when testing on and porting to embedded systems with no filesystem.

2.6.4.30 USE_CERT_BUFFERS_4096 Enables 4096-bit test certificate and key buffers located in `<wolfssl_root>/wolfssl/certs_test.h`. Helpful when testing on and porting to embedded systems with no filesystem.

2.6.4.31 USE_CERT_BUFFERS_25519 Enables Ed25519 test certificate and key buffers located in `<wolfssl_root>/wolfssl/certs_test.h`. Helpful when testing on and porting to embedded systems with no filesystem.

2.6.4.32 USE_WOLFSSL_IO This macro enables callbacks to send/recv. You can find an example of it in use here: (https://github.com/wolfSSL/wolfssl-examples/blob/master/tls/client-tls-callback.c#L6)

2.6.4.33 CUSTOM_RAND_GENERATE_SEED Allows user to define custom function equivalent to `wc_GenerateSeed(byte* output, word32 sz)`.

2.6.4.34 CUSTOM_RAND_GENERATE_BLOCK Allows user to define custom random number generation function. Examples of use are as follows.

```
./configure --disable-hashdrbg
CFLAGS="-DCUSTOM_RAND_GENERATE_BLOCK= custom_rand_generate_block"
```

Or

```
/* RNG */
/* #define HAVE_HASHDRBG */
extern int custom_rand_generate_block(unsigned char* output, unsigned int sz);
```

2.6.4.35 WC_NO_RNG Disables all RNG (Random Number Generator) support. Use when the application provides its own random data or when no randomness is needed (e.g., deterministic operations only).

2.6.4.36 HAVE_HASHDRBG Enables the Hash-based Deterministic Random Bit Generator (DRBG) per NIST SP 800-90A. This is the default RNG implementation in wolfSSL using SHA-256 as the underlying hash.

2.6.4.37 WC_RNG_BLOCKING Makes RNG operations blocking, retrying on transient failures rather than returning an error. Useful on platforms where the entropy source may temporarily be unavailable.

2.6.4.38 WC_VERBOSE_RNG Enables verbose debug output for RNG operations. Prints detailed information about seed generation, DRBG state, and health test results.

2.6.4.39 WC_X25519_NONBLOCK Enables non-blocking X25519 key agreement operations. Allows X25519 computations to yield and resume, useful for cooperative multitasking.

2.6.4.40 WC_RNG_SEED_CB Enables a custom seed callback function for the DRBG. Allows the application to provide its own entropy source via `wc_SetSeed_Cb()`.

2.6.4.41 WC_RNG_BANK_SUPPORT Enables RNG bank support for pre-generating random data. Allows buffering random bytes in advance for faster subsequent random number requests.

2.6.4.42 WOLFSSL_RNG_USE_FULL_SEED Uses the full seed length (384 bits) for DRBG seeding instead of the minimum required. Provides additional entropy margin.

2.6.4.43 WOLFSSL_GENSEED_FORTEST Uses a deterministic seed source for testing purposes. WARNING: This produces predictable random output and must never be used in production.

2.6.4.44 WOLFSSL_KEEP_RNG_SEED_FD_OPEN Keeps the `/dev/random` or `/dev/urandom` file descriptor open between seed operations instead of opening and closing it each time. Reduces overhead on systems with frequent reseeding.

2.6.4.45 CUSTOM_RAND_GENERATE Allows the user to define a custom random word generator function. The function should return a single random word (`unsigned int`).

2.6.4.46 CUSTOM_RAND_GENERATE_SEED_OS Allows the user to define a custom OS-level seed generator function, replacing the default platform-specific `GenerateSeed()` while still using the wolfSSL DRBG on top.

2.6.4.47 HAVE_ENTROPY_MEMUSE Enables the memory-use based entropy source. This entropy source measures timing variations in memory access patterns (cache hits/misses) to generate entropy for DRBG seeding.

2.6.4.48 ENTROPY_MEMUSE_FORCE_FAILURE Forces the memory-use entropy source to fail. Used for testing error handling paths in the entropy collection code.

2.6.4.49 HAVE_GETRANDOM Indicates that the Linux `getrandom()` syscall is available for entropy collection. Automatically detected on supported platforms.

2.6.4.50 WOLFSSL_GETRANDOM Enables use of the `getrandom()` syscall as the entropy source for DRBG seeding on Linux systems. More reliable than reading from `/dev/urandom` as it blocks until sufficient entropy is available.

2.6.4.51 FORCE_FAILURE_GETRANDOM Forces the `getrandom()` syscall to fail. Used for testing fallback entropy source paths.

2.6.4.52 NO_DEV_URANDOM Disables use of `/dev/urandom` for random seeding. When defined along with `NO_DEV_RANDOM`, an alternative seed source must be provided.

2.6.4.53 HAVE_AMD_RDSEED Enables use of AMD's RDSEED instruction for direct hardware entropy. Similar to `HAVE_INTEL_RDSEED` but for AMD processors.

2.6.4.54 IDIRECT_DEV_RANDOM Specifies a custom path for the random device on iDirect platforms instead of the default `/dev/random`.

2.6.4.55 WIN_REUSE_CRYPT_HANDLE Reuses the Windows `CryptContext` handle between random number generation calls instead of acquiring and releasing it each time. Improves performance on Windows.

2.6.4.56 WC_RNG_SEED_APT_CUTOFF Sets the cutoff value for the DRBG Adaptive Proportion Test (APT). The APT detects degradation in the entropy source by checking if any single value appears too frequently within a window.

2.6.4.57 WC_RNG_SEED_APT_WINDOW Sets the window size for the DRBG Adaptive Proportion Test (APT). Defines how many samples are examined in each test window.

2.6.4.58 WC_RNG_SEED_RCT_CUTOFF Sets the cutoff value for the DRBG Repetition Count Test (RCT). The RCT detects catastrophic entropy source failure by checking for consecutive identical outputs.

2.6.4.59 STM32_RNG Enables the STM32 hardware Random Number Generator peripheral for entropy collection.

2.6.4.60 STM32_NUTTX_RNG Enables STM32 hardware RNG access through the NuttX RTOS `/dev/random` interface.

2.6.4.61 WOLFSSL_STM32F427_RNG Enables hardware RNG support specific to the STM32F427 microcontroller.

2.6.4.62 WOLFSSL_STM32_RNG_NOLIB Enables direct register access to the STM32 RNG peripheral without using the STM32 HAL library. Useful for bare-metal deployments.

2.6.4.63 WOLFSSL_PIC32MZ_RNG Enables the Microchip PIC32MZ hardware Random Number Generator for entropy collection.

2.6.4.64 FREESCALE_RNGA Enables the Freescale/NXP RNGA (Random Number Generator Accelerator) hardware peripheral.

2.6.4.65 FREESCALE_K70_RNGA Enables RNGA support specific to the Freescale/NXP Kinetis K70 microcontroller family.

2.6.4.66 FREESCALE_RNGB Enables the Freescale/NXP RNGB (Random Number Generator version B) hardware peripheral.

2.6.4.67 FREESCALE_KSDK_2_0_RNGA Enables Freescale/NXP RNGA through the KSDK 2.0 SDK driver interface.

2.6.4.68 FREESCALE_KSDK_2_0_TRNG Enables Freescale/NXP TRNG (True Random Number Generator) through the KSDK 2.0 SDK driver interface.

2.6.4.69 MAX3266X_RNG Enables the Maxim MAX3266X hardware Random Number Generator.

2.6.4.70 QAT_ENABLE_RNG Enables hardware RNG through the Intel QuickAssist Technology (QAT) accelerator.

2.6.4.71 WOLFSSL_ATECC_RNG Enables hardware RNG from the Microchip ATECC508A/ATECC608A secure element.

2.6.4.72 WOLFSSL_SILABS_TRNG Enables the Silicon Labs True Random Number Generator (TRNG) for entropy collection.

2.6.4.73 WOLFSSL_SCE_NO_TRNG Disables the TRNG on Renesas Secure Crypto Engine (SCE). AES and other SCE features remain available but RNG uses a software implementation.

2.6.4.74 WOLFSSL_SCE_TRNG_HANDLE Specifies the Renesas SCE TRNG handle to use for random number generation.

2.6.4.75 WOLFSSL_SE050_NO_TRNG Disables the TRNG on NXP SE050 secure element. Other SE050 crypto operations remain available.

2.6.4.76 WOLFSSL_PSA_NO_RNG Disables RNG through the Platform Security Architecture (PSA) crypto API. Use when PSA is enabled but RNG should use a different source.

2.6.4.77 HAVE_IOTSAFE_HWRNG Enables hardware RNG from an IoT-Safe compliant SIM card or secure element.

2.6.4.78 WOLFSSL_XILINX_CRYPT_VERSAL Enables crypto hardware support on Xilinx Versal platforms, including the hardware TRNG for entropy collection.

COPYRIGHT ©2024 wolfSSL Inc.

2.6.4.79 NO_PUBLIC_GCM_SET_IV Use this if you have done your own custom hardware port and not provided a public implementation of `wc_AesGcmSetIV()`

2.6.4.80 NO_PUBLIC_CCM_SET_NONCE Use this if you have done your own custom hardware port and not provided a public implementation of `wc_AesGcmSetNonce()`

2.6.4.81 NO_GCM_ENCRYPT_EXTRA Use this if you have done your own custom hardwareport and not provided an implementation of `wc_AesGcmEncrypt_ex()`

2.6.4.82 WOLFSSL_STM32[F1 | F2 | F4 | F7 | L4] Use one of these defines when building for the appropriate STM32 device. Update `wolfssl-root/wolfssl/wolfcrypt/settings.h` section with regards to the wolfSSL porting guide (https://www.wolfssl.com/docs/porting-guide/) as appropriate.

2.6.4.83 WOLFSSL_STM32_CUBEMX When using the CubeMX tool to generate Hardware Abstraction Layer (HAL) API's use this setting to add appropriate support in wolfSSL.

2.6.4.84 WOLFSSL_CUBEMX_USE_LL When using the CubeMX tool to generate APIs there are two options, HAL (Hardware Abstraction Layer) or Low Layer (LL). Use this define to control which headers are include in `wolfssl-root/wolfssl/wolfcrypt/settings.h` in the `WOLFSSL_STM32[F1/F2/F4/F7/L4]` section.

2.6.4.85 NO_STM32_CRYPTO For when an STM32 part does not offer hardware crypto support

2.6.4.86 NO_STM32_HASH For when an STM32 part does not offer hardware hash support

2.6.4.87 NO_STM32_RNG For when an STM32 part does not offer hardware RNG support

2.6.4.88 XTIME_MS Macro to map a function for use to get the time in milliseconds when using TLS 1.3. Example being:

```
extern time_t m2mb_xtime_ms(time_t * timer);
#define XTIME_MS(tl) m2mb_xtime_ms((tl))
```

2.6.4.89 WOLFSSL_CIPHER_TEXT_CHECK Define this to check for possible glitching attack against the AES encrypt operation during a TLS connection.

2.6.4.90 RTTHREAD RT-THREAD macro is used when porting rtthread IoT to wolfSSL.

2.6.4.91 SO_REUSEPORT Allows local address and port reuse.

2.6.4.92 INTIME_RTOS Port setting for INtime RTOS.

2.6.4.93 WOLFSSL_SGX Used when porting to SGX.

2.6.5 Reducing Memory or Code Usage

2.6.5.1 TFM_TIMING_RESISTANT Can be defined when using fast math (`USE_FAST_MATH`) on systems with a small stack size. This will get rid of the large static arrays.

2.6.5.2 ECC_TIMING_RESISTANT This is used as a Timing Resistance feature that enables code in ecc.c to prevent side channel and differential power analysis (DPA) attacks.

2.6.5.3 FUSION_RTOS A Fusion RTOS implementation is used for tickets to represent the difference between when they are first seen and when they are sent. It returns the time in milliseconds as a 32-bit value.

2.6.5.4 WOLFSSL_SMALL_STACK Can be used for devices which have a small stack size. This increases the use of dynamic memory in `wolfcrypt/src/integer.c`, but can lead to slower performance.

2.6.5.5 WOLFSSL_PTHREADS Use pthread-based mutex and threading implementations. Autodetected on most POSIX systems.

2.6.5.6 WOLFSSL_MUTEX_INITIALIZER Use static mutex initialization (e.g., `PTHREAD_MUTEX_INITIALIZER`) instead of dynamic `pthread_mutex_init`. Useful for reducing initialization overhead.

2.6.5.7 WC_MUTEX_OPS_INLINE Use inlined mutex operations instead of function calls. Can improve performance on platforms where mutex operations are frequent.

2.6.5.8 WOLFSSL_USE_RWLOCK Enable reader-writer lock support for improved concurrency in read-heavy workloads.

2.6.5.9 WOLFSSL_THREAD_NO_JOIN Create threads without join capability (detached threads). Useful on platforms that do not support thread joining.

2.6.5.10 WOLFSSL_THREADED_CRYPT Enables multi-threaded cryptographic operations for improved performance on multi-core systems. Allows encryption/decryption to run in parallel.

2.6.5.11 WOLFSSL_ALGO_HW_MUTEX Enable per-algorithm hardware mutex locks for AES, hash, public-key, and RNG operations. Useful when hardware crypto engines require serialized access.

2.6.5.12 WOLFSSL_CRYPT_HW_MUTEX Master control for hardware crypto mutex initialization. When enabled, provides `wolfSSL_CryptHwMutexInit`, `Lock`, and `UnLock` functions.

2.6.5.13 USE_WOLFSSL_MEMORY Enable custom memory allocation hooks (`wolfSSL_SetAllocators`). On by default. Allows replacing malloc/realloc/free with custom implementations.

2.6.5.14 WOLFSSL_TRACK_MEMORY Enable memory allocation tracking and statistics. Useful for profiling memory usage in wolfSSL/wolfCrypt.

2.6.5.15 WOLFSSL_TRACK_MEMORY_VERBOSE Enable verbose memory tracking output with per-allocation details. Extends `WOLFSSL_TRACK_MEMORY`.

2.6.5.16 WOLFSSL_MEM_FAIL_COUNT Count malloc failures for testing. Allows testing error handling paths by failing after a specified number of allocations.

2.6.5.17 WOLFSSL_CHECK_MEM_ZERO Verify that sensitive memory (key material, etc.) is properly zeroed on free. Debug tool for detecting potential key material leaks.

2.6.5.18 WOLFSSL_GMTIME Provide a custom gmtime implementation for platforms without standard C library time functions.

2.6.5.19 STRING_USER User provides all string function implementations. Disables built-in string function wrappers.

2.6.5.20 USE_WOLF_STRTOK Use wolfSSL's built-in strtok implementation for portability on platforms without strtok_r.

2.6.5.21 USE_WOLF_STRSEP Use wolfSSL's built-in strsep implementation for portability.

2.6.5.22 USE_WOLF_STRLCPY Use wolfSSL's built-in strlcpy implementation for portability on platforms without BSD strlcpy.

2.6.5.23 USE_WOLF_STRLCAT Use wolfSSL's built-in strlcat implementation for portability.

2.6.5.24 USE_WOLF_STRCASECMP Use wolfSSL's built-in case-insensitive string comparison for portability.

2.6.5.25 USE_WOLF_STRNCASECMP Use wolfSSL's built-in length-limited case-insensitive string comparison for portability.

2.6.5.26 USE_WOLF_STRDUP Use wolfSSL's built-in strdup implementation for portability.

2.6.5.27 WOLFSSL_ATOMIC_OPS Enable atomic operations for thread-safe reference counting and other operations without requiring full mutexes.

2.6.5.28 WOLFSSL_ATOMIC_INITIALIZER Provides a static initializer for atomic variables used in thread-safe cleanup operations.

2.6.5.29 WOLFSSL_USER_DEFINED_ATOMICS User-provided atomic operation implementations. Define this when the platform requires custom atomic primitives.

2.6.5.30 WOLFSSL_LEANPSK Lean PSK (Pre-Shared Key) build with minimal features. Reduces code size by disabling non-essential features.

2.6.5.31 WOLF_C89 Enable C89 compatibility mode. Ensures the codebase compiles with strict C89/ANSI C compilers.

2.6.5.32 ALT_ECC_SIZE If using fast math and RSA/DH you can define this to reduce your ECC memory consumption. Instead of using stack for ECC points it will allocate from the heap.

2.6.5.33 ECC_SHAMIR Uses variation of ECC math that is slightly faster, but doubles heap usage.

COPYRIGHT ©2024 wolfSSL Inc.

2.6.5.34 RSA_LOW_MEM When defined CRT is not used which saves on some memory but slows down RSA operations. It is off by default.

2.6.5.35 WOLFSSL_SHA3_SMALL When SHA3 is enabled this macro will reduce build size.

2.6.5.36 WOLFSSL_SHUTDOWNONCE Ensures `wolfSSL_shutdown()` only sends one `close_notify` alert even if called multiple times. Prevents duplicate shutdown messages.

2.6.5.37 WOLFSSL_SHAKE128 Enables SHAKE128 extendable-output function (XOF) based on SHA-3. Provides variable-length output.

2.6.5.38 WOLFSSL_SHAKE256 Enables SHAKE256 extendable-output function (XOF) based on SHA-3. Provides variable-length output.

2.6.5.39 SHA3_BY_SPEC Uses the specification-ordered Keccak-f permutation. By default, wolfSSL uses an optimized bit-interleaved ordering.

2.6.5.40 WC_SHA3_NO_ASM Disables assembly-optimized SHA-3 implementation. Forces use of the portable C implementation.

2.6.5.41 WC_SHA3_FAULT_HARDEN Hardens SHA-3 against fault injection attacks by performing redundant computations and verifying consistency.

2.6.5.42 WC_ASYNC_ENABLE_SHA3 Enables asynchronous SHA-3 operations via the wolfSSL async crypto framework.

2.6.5.43 STM32_HASH_SHA3 Enables STM32 hardware SHA-3 acceleration.

2.6.5.44 PSOC6_HASH_SHA3 Enables Cypress/Infineon PSoC6 hardware SHA-3 acceleration.

2.6.5.45 WOLFSSL_SMALL_CERT_VERIFY Verify the certificate signature without using DecodedCert. Doubles up on some code but allows smaller peak heap memory usage. Cannot be used with `WOLFSSL_NONBLOCK_OCSP`.

2.6.5.46 GCM_SMALL Option to reduce AES GCM code size by calculating at runtime instead of using tables. Possible options are: `GCM_SMALL`, `GCM_WORD32` or `GCM_TABLE`.

2.6.5.47 CURVED25519_SMALL Defines `CURVE25519_SMALL` and `ED25519_SMALL`.

2.6.5.48 CURVE25519_SMALL Use small memory option for curve25519. This uses less memory, but is slower.

2.6.5.49 ED25519_SMALL Use small memory option for ed25519. This uses less memory, but is slower.

2.6.5.50 USE_SLOW_SHA Reduces code size by not unrolling loops, which reduces performance for SHA.

2.6.5.51 WC_HASH_DATA_ALIGNMENT Specifies required data alignment for hash input. Some hardware backends require aligned input buffers.

2.6.5.52 WC_ASYNC_ENABLE_SHA Enables asynchronous SHA-1 operations via the wolfSSL async crypto framework.

2.6.5.53 WOLFSSL_PIC32MZ_HASH Enables Microchip PIC32MZ hardware hash acceleration for SHA-1 and SHA-256.

2.6.5.54 WOLFSSL_TI_HASH Enables Texas Instruments hardware hash acceleration.

2.6.5.55 WOLFSSL_RENESAS_RX64_HASH Enables Renesas RX64 hardware hash acceleration.

2.6.5.56 FREESCALE_LTC_SHA Enables Freescale/NXP LTC (Low Power Trusted Cryptography) SHA acceleration.

2.6.5.57 FREESCALE_MMCAU_SHA Enables Freescale/NXP MMCAU (Memory-Mapped Cryptographic Acceleration Unit) SHA acceleration.

2.6.5.58 PSOC6_HASH_SHA1 Enables Cypress/Infineon PSoC6 hardware SHA-1 acceleration.

2.6.5.59 USE_SLOW_SHA256 Reduces code size by not unrolling loops, which reduces performance for SHA. About 2k smaller and about 25% slower.

2.6.5.60 USE_SLOW_SHA512 Reduces code size by not unrolling loops, which reduces performance for SHA. Over twice as small, but 50% slower.

2.6.5.61 WOLFSSL_NOSHA512_224 Disables the SHA-512/224 variant. Reduces code size when SHA-512/224 is not needed.

2.6.5.62 WOLFSSL_NOSHA512_256 Disables the SHA-512/256 variant. Reduces code size when SHA-512/256 is not needed.

2.6.5.63 USE_SLOW_SHA2 Disables loop unrolling for all SHA-2 family algorithms. Reduces code size at the cost of performance.

2.6.5.64 WOLFSSL_HASH_FLAGS Enables hash flags for tracking hash state (e.g., whether the hash has been finalized). Used by some hardware backends.

2.6.5.65 WOLFSSL_HASH_KEEP Keeps the hash input data in memory for potential reuse. Allows re-hashing of data without re-feeding it.

2.6.5.66 WC_ASYNC_ENABLE_SHA512 Enables asynchronous SHA-512 operations via the wolfSSL async crypto framework.

2.6.5.67 WC_ASYNC_ENABLE_SHA384 Enables asynchronous SHA-384 operations via the wolfSSL async crypto framework.

2.6.5.68 WOLFSSL_KCAPI_HASH Enables hash operations through the Linux kernel crypto API (AF_ALG). Offloads SHA and other hashes to the kernel.

2.6.5.69 WOLFSSL_SE050_HASH Enables hash acceleration on the NXP SE050 secure element.

2.6.5.70 WOLFSSL_SILABS_SHA384 Enables Silicon Labs hardware acceleration for SHA-384.

2.6.5.71 WOLFSSL_SILABS_SHA512 Enables Silicon Labs hardware acceleration for SHA-512.

2.6.5.72 WOLFSSL_ARMASM_CRYPTO_SHA512 Enables ARM crypto extension instructions for SHA-512 acceleration.

2.6.5.73 WOLFSSL_RENESAS_RSIP Enables Renesas RSIP (Renesas Security IP) hardware acceleration for hashing and crypto operations.

2.6.5.74 ECC_USER_CURVES Allow user to choose ECC curve sizes that are enabled. Only the 256-bit curve is enabled by default. To enable others use HAVE_ECC192, HAVE_ECC224, etc...

2.6.5.75 WOLFSSL_SP_NO_MALLOC In the SP code, always use stack, no heap XMALLOC()/XREALLOC()/XFREE() calls are made.

2.6.5.76 WOLFSSL_SP_NO_DYN_STACK Disable use of dynamic stack items. Used with small code size and not small stack.

2.6.5.77 WOLFSSL_SP_FAST_MODEXP Compiles in a faster mod_exp implementation at the expense of code size.

2.6.5.78 WC_DISABLE_RADIX_ZERO_PAD Disable printing of leading zero in hexadecimal string output. For example, if this macro is defined, the value 8 will be printed as the string “0x8” but if it is not defined it will be printed as “0x08”. Defining this macro can reduce code size.

2.6.5.79 WC_ASN_NAME_MAX This allows overriding the maximum name support for an X.509 certificate field.

2.6.5.80 OPENSSL_EXTRA_X509_SMALL Special small OpenSSL compat layer for certs.

2.6.5.81 OPENSSL_EXTRA_NO_ASN1 Enables OpenSSL extra compatibility APIs but excludes ASN1 object functions. Useful when ASN1 compatibility is not needed to reduce code size.

2.6.6 Increasing Performance

2.6.6.1 USE_INTEL_SPEEDUP Enables use of Intel’s AVX/AVX2 instructions for accelerating AES, ChaCha20, Poly1305, SHA256, SHA512, ED25519 and Curve25519.

2.6.6.2 WOLFSSL_AESNI Enables use of AES accelerated operations which are built into some Intel and AMD chipsets. When using this define, the `aes_asm.asm` (for Windows with at&t syntax) or `aes_asm.S` file is used to optimize via the Intel AES new instruction set (AESNI).

2.6.6.3 WOLFSSL_AESNI_BY4 Enables 4-block parallel AES-NI processing. Processes four AES blocks simultaneously using AES-NI pipelining for improved throughput. Requires WOLFSSL_AESNI.

2.6.6.4 WOLFSSL_AESNI_BY6 Enables 6-block parallel AES-NI processing. Processes six AES blocks simultaneously using AES-NI pipelining for maximum throughput. Requires WOLFSSL_AESNI.

2.6.6.5 WOLFSSL_AES_SMALL_TABLES Uses smaller AES S-box lookup tables. Reduces code/data size at the cost of slightly slower AES operations. Useful for memory-constrained embedded targets.

2.6.6.6 WOLFSSL_AES_NO_UNROLL Disables loop unrolling in AES round functions. Reduces code size at the cost of performance. Useful for constrained environments where code size matters more than speed.

2.6.6.7 WOLFSSL_AES_TOUCH_LINES Touch all AES table cache lines before lookups to provide side-channel resistance. Mitigates cache-timing attacks by ensuring all table entries are in cache before use.

2.6.6.8 WC_AES_BITSLICED Enables bitsliced AES implementation. Uses a bitwise-parallel technique that processes multiple blocks simultaneously and provides constant-time execution for side-channel resistance.

2.6.6.9 AES_GCM_GMULT_NCT Enables non-constant-time GCM GMULT implementation. Faster but not protected against cache-timing side-channel attacks. Only use when side-channel resistance is not required.

2.6.6.10 NO_WOLFSSL_ALLOC_ALIGN Disables aligned memory allocation for AES contexts. By default, AES contexts are aligned to cache line boundaries for performance. Disable on platforms that do not support aligned allocation.

2.6.6.11 WC_ASYNC_ENABLE_AES Enables asynchronous AES operations. Allows AES encrypt/decrypt to be offloaded to hardware accelerators using the wolfSSL async crypto framework.

2.6.6.12 WOLFSSL_CRYPTOCELL_AES Enables AES acceleration using ARM CryptoCell hardware. Requires the CryptoCell SDK and WOLFSSL_CRYPTOCELL.

2.6.6.13 WOLFSSL_DEVCRYPTO_AES Enables AES acceleration via Linux `/dev/crypto` interface. Requires WOLFSSL_DEVCRYPTO.

2.6.6.14 WOLFSSL_DEVCRYPTO_CBC Enables AES-CBC acceleration via Linux `/dev/crypto` interface. Requires WOLFSSL_DEVCRYPTO.

2.6.6.15 WOLFSSL_KCAPI_AES Enables AES operations through the Linux kernel crypto API (AF_ALG). Offloads AES to the kernel's crypto subsystem.

2.6.6.16 WOLFSSL_NO_KCAPI_AES_CBC Disables AES-CBC through KCAPI when WOLFSSL_KCAPI_AES is enabled. Useful when only non-CBC AES modes are needed through the kernel crypto API.

2.6.6.17 WOLFSSL_PSA_NO_AES Disables AES through the Platform Security Architecture (PSA) crypto API. Use when PSA is enabled but AES should use the software implementation instead.

2.6.6.18 WOLFSSL_SCE_NO_AES Disables AES through the Renesas Secure Crypto Engine (SCE). Use when SCE is enabled but AES should use the software implementation.

2.6.6.19 NO_IMX6_CAAM_AES Disables AES acceleration on NXP i.MX6 CAAM (Cryptographic Acceleration and Assurance Module). Use when CAAM is enabled but AES should use the software implementation.

2.6.6.20 WOLFSSL_AFALG_XILINX_AES Enables AES acceleration through AF_ALG on Xilinx platforms. Uses the Xilinx crypto hardware via the Linux AF_ALG interface.

2.6.6.21 NO_WOLFSSL_ESP32_CRYPT_AES Disables ESP32 hardware AES acceleration. Use when building for ESP32 but AES should use the software implementation.

2.6.6.22 STM32_CRYPTO_AES_ONLY Restricts STM32 hardware crypto to AES operations only. Other algorithms will use software implementations even when STM32 crypto hardware is available.

2.6.6.23 WC_DEBUG_CIPHER_LIFECYCLE Enables debug logging for AES cipher context lifecycle events (init, set key, free). Useful for debugging resource leaks or double-free issues with AES contexts.

2.6.6.24 WOLFSSL_HW_METRICS Enables tracking of hardware acceleration usage metrics. When enabled, wolfSSL counts how many operations were offloaded to hardware vs. handled in software, accessible via `wolfCrypt_GetHwMetrics()`.

2.6.6.25 HAVE_INTEL_RDSEED Enable Intel's RDSEED for DRBG seed source.

2.6.6.26 HAVE_INTEL_RDRAND Enable Intel's RDRAND instruction for wolfSSL's random source.

2.6.6.27 FP_ECC Enables ECC Fixed Point Cache, which speeds up repeated operations against same private key. Can also define number of entries and LUT bits using FP_ENTRIES and FP_LUT to reduce default static memory usage.

2.6.6.28 FP_ENTRIES Defines the number of cache entries (default 15) for the ECC fixed-point multiplication lookup table. Requires FP_ECC. Adjust to balance memory usage and performance.

2.6.6.29 FP_LUT Sets the lookup table bit size (2-12, default 8) for ECC fixed-point precomputation. Larger values use more memory but provide faster verification. Requires FP_ECC.

2.6.6.30 FP_ECC_CONTROL Auto-selects cached fixed-point ECC verification using SP functions when WOLFSSL_HAVE_SP_ECC is available. Enabled by default when applicable.

2.6.6.31 HAVE_ECC_CHECK_PUBKEY_ORDER Enables ECC public key order validation during import to detect invalid keys. Auto-enabled unless NO_ECC_CHECK_PUBKEY_ORDER is defined or hardware accelerators are in use.

2.6.6.32 HAVE_ECC_MAKE_PUB Enables the `wc_ecc_make_pub` function to compute a public key from a private key. Enabled by default.

2.6.6.33 HAVE_ECC_VERIFY_HELPER Enables ECC signature verification helper functions. Auto-enabled unless hardware accelerators are in use.

2.6.6.34 NO_ECC_CHECK_PUBKEY_ORDER Disables ECC public key order validation checks during key import. Not recommended for production use as it skips important security validation.

2.6.6.35 WC_NO_CACHE_RESISTANT Disables cache-resistant operations (conditional swaps) in ECC scalar multiplication to reduce overhead. Not recommended as it may expose operations to cache-based side-channel attacks.

2.6.6.36 WOLFSSL_ECC_NO_SMALL_STACK Disables `WOLFSSL_SMALL_STACK` optimizations for ECC operations, forcing stack allocation instead of heap. Useful when stack space is plentiful and heap allocation overhead is undesirable.

2.6.6.37 WOLFSSL_PUBLIC_ECC_ADD_DBL Makes `ecc_projective_add_point` and `ecc_projective_dbl_poi` public APIs instead of internal-only functions. Useful for applications that need direct access to ECC point arithmetic.

2.6.6.38 WOLFSSL_PYTHON Enables APIs and behaviors needed for the Python wolfSSL module compatibility.

2.6.6.39 SQRTMOD_USE_MOD_EXP Computes square root modulo prime using modular exponentiation instead of the Jacobi symbol method for compressed key decompression. Off by default.

2.6.6.40 WOLFSSL_ECIES_OLD Uses the original wolfSSL ECIES format where the public key is not included in the shared secret material. Off by default.

2.6.6.41 WOLFSSL_ECDSA_MATCH_HASH Requires the ECDSA signature hash algorithm to match the curve's preferred hash (e.g., P-256 uses SHA-256, P-384 uses SHA-384).

2.6.6.42 WOLFSSL_ECIES_ISO18033 Uses the ISO 18033 ECIES standard which includes the public key in the shared secret derivation. Off by default.

2.6.6.43 WOLFSSL_ECIES_GEN_IV Generates a random IV for ECIES encryption instead of deriving it from the KDF output. Off by default.

2.6.6.44 WOLFSSL_SP_521 Enables the single-precision (SP) math optimized implementation for the P-521 ECC curve. Off by default. Enabled by `--enable-sp` and its variants (for example `--enable-sp=521`, `ec521`, `p521`) on 64-bit Intel/ARM targets, by the CMake equivalent, or by defining it manually in `user_settings.h`. It is **not** auto-enabled by the headers from `WOLFSSL_SP_MATH` / `WOLFSSL_SP_MATH_ALL` together with `HAVE_ECC521`; defining `WOLFSSL_SP_521` instead forces `HAVE_ECC521` on. Note that `WOLFSSL_SP_MATH_ALL` alone still provides working (unoptimized, generic) P-521 support; `WOLFSSL_SP_521` adds the curve-specific optimized path.

2.6.6.45 WOLFSSL_SP_SM2 Enables single-precision (SP) math optimized implementation for the SM2 curve (Chinese cryptographic standard). Auto-enabled when `WOLFSSL_SM2` is set.

2.6.6.46 WOLF_CRYPTO_CB_ONLY_ECC Restricts ECC operations to use only crypto callbacks, disabling all software ECC implementations. Useful when all ECC operations should be delegated to hardware or an external module. Off by default.

2.6.6.47 WC_ASYNC_ENABLE_ECC Enables asynchronous (non-blocking) ECC operations with crypto callbacks. Requires WOLFSSL_ASYNC_CRYPT. Off by default.

2.6.6.48 WC_ASYNC_ENABLE_ECC_KEYGEN Enables asynchronous ECC key generation, allowing key generation to be offloaded to hardware accelerators. Requires WOLFSSL_ASYNC_CRYPT. Off by default.

2.6.6.49 PLUTON_CRYPTO_ECC Enables use of ARM Pluton trusted execution environment for ECC operations. Off by default.

2.6.6.50 WOLFSSL_CAAM_BLACK_KEY_SM Uses NXP CAAM secure memory for encrypted black key storage in ECC operations. Off by default.

2.6.6.51 WOLFSSL_KCAPI_ECC Offloads ECC operations to the Linux Kernel Crypto API (kcAPI) for hardware acceleration. Off by default.

2.6.6.52 WOLFSSL_ASYNC_CRYPT This enables support for asynchronous cryptography using hardware based adapters such as the Intel QuickAssist or Marvell (Cavium) Nitrox V. The asynchronous code is not included in the public distribution and is available for evaluation by contacting us via email at facts@wolfssl.com.

2.6.6.53 WOLFSSL_NO_ASYNC_IO This disables the asynchronous I/O networking. Asynchronous I/O is on by default and can take up to around 140 bytes during the handshaking process. If your network interface doesn't return SOCKET_EWOULDBLOCK or SOCKET_EAGAIN (or WOLFSSL_CBIO_ERR_WANT_WRITE for custom I/O callbacks) on writing you can define WOLFSSL_NO_ASYNC_IO to have wolfSSL not save the state while building handshake messages.

2.6.7 GCM Performance Tuning

There are 4 variants of GCM performance:

- GCM_SMALL - Smallest footprint, slowest (FIPS validated)
- GCM_WORD32 - Medium (FIPS validated)
- GCM_TABLE - Fast (FIPS validated)
- GCM_TABLE_4BIT - Fastest (FIPS validated)

2.6.8 wolfSSL's Math Options

There are three math libraries in wolfSSL.

- Big Integer
- Fast Math
- Single Precision Math

When building wolfSSL, only one of these must be used.

Big Integer Library is the most portable option as it is written in C without any assembly. As such it is not optimized for specific architectures. All math variables are instantiated on the heap; minimal stack usage. Unfortunately, Big Integer Library is not timing resistant.

Fast Math Library is a good option. It is implemented using both C and assembly. As such, it has optimizations for specific architectures. All math variables are instantiated on the stack, leading to less heap usage. It can be made timing resistant if the TFM_TIMING_RESISTANT macro is defined. We have taken it through FIPS 140-2 and 140-3 certifications.

Single Precision (SP) Math Library is our recommended library. It is implemented using both C and assembly. As such, it has optimizations for specific architectures. All math variables are instantiated on the stack, leading to less heap usage. It is always timing resistant. It is generally optimized for speed at the cost of code size, but is highly configurable to compile out unneeded code. We have taken it through DO-178C certifications.

2.6.8.1 Big Integer Math Library (Deprecation Planned) This library is planned to be deprecated and removed from the wolfSSL/wolfCrypt library by the end of 2023. If desired this can be enabled with `--enable-heapmath` or `CFLAGS=-DUSE_INTEGER_HEAP_MATH`.

Forked from public domain LibTomMath library. For more information about LibTomMath, please see https://www.libtom.net/LibTomMath/. Please note that our fork is considerably more active and secure than the original public domain code.

This library is generally the most portable and easiest to get going with. The negatives to the normal big integer library are that it is slower, it uses a lot of heap memory as all memory is allocated from the heap, requires an `XREALLOC()` implementation, and is not timing resistant. The implementation can be found in `integer.c`.

2.6.8.2 Fast Math

2.6.8.2.1 USE_FAST_MATH Forked from public domain LibTomFastMath library. For more information about LibTomFastMath, please see https://www.libtom.net/TomsFastMath. Please note that our fork is considerably more active and secure than the original public domain code from LibTomFastMath. We have improved performance, security and code quality. Also we have taken the FastMath code through FIPS 140-2 and 140-3 certifications.

The FastMath library uses assembly if possible, and will speed up asymmetric private/public key operations like RSA, DH, and DSA. Assembly inclusion is dependent on compiler and processor combinations. Some combinations will need additional configure flags and some may not be possible. Help with optimizing FastMath with new assembly routines is available on a consulting basis. See the Architecture-Specific Optimizations.

For FastMath, all memory is allocated on the stack. Because the stack memory usage can be larger when using FastMath, we recommend defining TFM_TIMING_RESISTANT as well when using this option. The FastMath code is timing resistant if TFM_TIMING_RESISTANT is defined. This will reduce some of the large math windows for constant time, which use less memory. This uses less stack because there are no shortcuts and therefore less branching during private key operations. This also makes the implementation more secure as timing attacks are a real threat and can give malicious third parties enough information to reproduce the private key.

On ia32, for example, all of the registers need to be available so high optimization and omitting the frame pointer needs to be taken care of. wolfSSL will add `-O3 -fomit-frame-pointer` to GCC for non debug builds. If you're using a different compiler you may need to add these manually to `CFLAGS` during configure.

OS X will also need `-mdynamic-no-pic` added to `CFLAGS`. In addition, if building in shared mode for ia32 on OS X you'll need to pass options to `LDFLAGS` as well:

```
LDFLAGS="-Wl,-read_only_relocs,warning"
```

This gives warnings for some symbols instead of errors.

FastMath also changes the way dynamic and stack memory are used. The normal math library uses dynamic memory for big integers. FastMath uses fixed size buffers that hold 4096 bit integers by default, allowing for 2048 bit by 2048 bit multiplications. If you need 4096 bit by 4096 bit multiplications then change FP_MAX_BITS in wolfssl/wolfcrypt/tfm.h. As FP_MAX_BITS is increased, this will also increase the runtime stack usage since the buffers used in the public key operations will now be larger. FP_MAX_BITS needs to be double the max key size. For example if your biggest key is 2048-bit, FP_MAX_BITS should be 4096 and if it is 4096-bit FP_MAX_BITS should be 8192. If using ECC only this can be reduced to the maximum ECC key size times two. A couple of functions in the library use several temporary big integers, meaning the stack can get relatively large. This should only come into play on embedded systems or in threaded environments where the stack size is set to a low value. If stack corruption occurs with FastMath during public key operations in those environments, increase the stack size to accommodate the stack usage.

If you are enabling FastMath without using the autoconf system, you'll need to define USE_FAST_MATH and add tfm.c to the wolfSSL build while removing integer.c. Defining ALT_ECC_SIZE will allocate ECC points only from the heap instead of the stack.

2.6.8.2.2 Architecture-Specific Optimizations The following macros can be defined for assembly optimizations with USE_FAST_MATH.

- TFM_ARM
- TFM_SSE2
- TFM_AVR32
- TFM_PPC32
- TFM_PPC64
- TFM_MIPS
- TFM_X86
- TFM_X86_64

If none of these are defined or TFM_NO_ASM is defined, then TFM_ISO will be defined and ISO C portable code will be used.

2.6.8.2.3 Algorithm-Specific Optimizations When enabled, optimized implementations for multiplication and squaring are used for the respective ECC curve.

- TFM_ECC192
- TFM_ECC224
- TFM_ECC256
- TFM_ECC384
- TFM_ECC521

2.6.8.2.4 TFM_SMALL_SET Speed optimization for multiplication of smaller numbers. Includes implementations of 1-16 word Comba multiplication and squaring. Useful for improving the performance of ECC operations.

2.6.8.2.5 TFM_HUGE_SET Speed optimization for multiplication of larger numbers. Includes implementations of 20, 24, 28, 32, 48 and 64 word Comba multiplication and squaring where bit size allows. Useful for improving the performance of RSA/DH/DSA operations.

2.6.8.2.6 TFM_SMALL_MONT_SET Speed optimization for montgomery reduction of smaller numbers on Intel architectures. Includes implementations of 1-16 word Montgomery reduction. Useful for improving the performance of ECC operations.

2.6.8.3 Proprietary Single Precision (SP) Math Support SP math is our recommended default option and has been taken through DO-178C certifications. Use these to speed up public key operations for specific keys sizes and curves that are common. Make sure to include the correct code files such as:

- `sp_c32.c`
- `sp_c64.c`
- `sp_arm32.c`
- `sp_arm64.c`
- `sp_armthumb.c`
- `sp_cortexm.c`
- `sp_int.c`
- `sp_x86_64.c`
- `sp_x86_64_asm.S`
- `sp_x86_64_asm.asm`

2.6.8.3.1 WOLFSSL_SP Enable Single Precision math support.

2.6.8.4 WOLFSSL_SP_MATH Enable only SP math and algorithms. Eliminates big integer math code such as normal (`integer.c`) or fast (`tfm.c`). Restricts key sizes and curves to only ones supported by SP.

2.6.8.5 WOLFSSL_SP_MATH_ALL Enable SP math and algorithms. Implements big integer math code such as normal (`integer.c`) or fast (`tfm.c`) for key sizes and curves not supported by SP.

2.6.8.6 WOLFSSL_SP_SMALL If using SP math this will use smaller versions of the code and avoid large stack variables.

2.6.8.6.1 SP_WORD_SIZE Force 32-bit or 64-bit data type for storing one word of a number.

2.6.8.6.2 WOLFSSL_SP_NONBLOCK Enables “non blocking” mode for Single Precision math, which will return FP_WOULDBLOCK for long operations and function must be called again until complete. Currently, this in only supported for ECC and is used in conjunction with `WC_ECC_NONBLOCK`.

2.6.8.6.3 WOLFSSL_SP_FAST_NCT_EXPTMOD Enables the faster non-constant time modular exponentiation implementation. Will only be used for public key operations; not private key operations.

2.6.8.6.4 WOLFSSL_SP_INT_NEGATIVE Allows multi-precision numbers to be negative. (Not required for cryptographic operations.)

2.6.8.6.5 WOLFSSL_SP_INT_DIGIT_ALIGN Enable when unaligned access of sp_int_digit pointer is not allowed.

2.6.8.6.6 WOLFSSL_HAVE_SP_RSA Single Precision RSA for 2048, 3072 and 4096 bit.

2.6.8.6.7 WOLFSSL_HAVE_SP_DH Single Precision DH for 2048, 3072 and 4096 bit.

2.6.8.6.8 WOLFSSL_HAVE_SP_ECC Single Precision ECC for SECP256R1 and SECP384R1.

2.6.8.6.9 WOLFSSL_SP_LARGE_CODE Allows Single-Precision (SP) speedups that come at the cost of larger binary size. Might not be appropriate for some embedded platforms.

2.6.8.6.10 WOLFSSL_SP_DIV_WORD_HALF Indicates that division using a double length word isn't available. For example, on 32-bit CPUs, if you do not want to compile in a 64-bit division from a library, then defining this macro turn on an implementation where the divide is done using half word size portions.

2.6.8.6.11 WOLFSSL_SP_DIV_32 Indicates that 32-bit division isn't available and that wolfSSL should use its own Single-Precision (SP) implementation.

2.6.8.6.12 WOLFSSL_SP_DIV_64 Indicates that 64-bit division isn't available and that wolfSSL should use its own Single-Precision (SP) implementation.

2.6.8.6.13 WOLFSSL_SP_ASM Enables Single-Precision (SP) platform specific assembly code implementation that is faster. Platform is detected.

2.6.8.6.14 WOLFSSL_SP_X86_64_ASM Enable Single-Precision (SP) Intel x64 assembly implementation.

2.6.8.6.15 WOLFSSL_SP_ARM32_ASM Enable Single-Precision (SP) Aarch32 assembly implementation.

2.6.8.6.16 WOLFSSL_SP_ARM64_ASM Enable Single-Precision (SP) Aarch64 assembly implementation.

2.6.8.6.17 WOLFSSL_SP_ARM_CORTEX_M_ASM Enable Single-Precision (SP) Cortex-M family (including Cortex-M4) assembly implementation.

2.6.8.6.18 WOLFSSL_SP_ARM_THUMB_ASM Enable Single-Precision (SP) ARM Thumb assembly implementation (used with -mthumb).

2.6.8.6.19 WOLFSSL_SP_X86_64 Enable Single-Precision (SP) Intel x86 64-bit assembly speedup macros. Only applies if `WOLFSSL_SP_MATH_ALL` is defined. See `sp_int.c`.

2.6.8.6.20 WOLFSSL_SP_X86 Enable Single-Precision (SP) Intel x86 assembly speedup macros. Only applies if `WOLFSSL_SP_MATH_ALL` is defined. See `sp_int.c`.

2.6.8.6.21 WOLFSSL_SP_PPC64 Enable Single-Precision (SP) PPC64 assembly speedup macros. Only applies if `WOLFSSL_SP_MATH_ALL` is defined. See `sp_int.c`.

2.6.8.6.22 WOLFSSL_SP_PPC Enable Single-Precision (SP) PPC assembly speedup macros. Only applies if `WOLFSSL_SP_MATH_ALL` is defined. See `sp_int.c`.

2.6.8.6.23 WOLFSSL_SP_MIPS64 Enable Single-Precision (SP) MIPS64 assembly speedup macros. Only applies if `WOLFSSL_SP_MATH_ALL` is defined. See `sp_int.c`.

2.6.8.6.24 WOLFSSL_SP_MIPS Enable Single-Precision (SP) MIPS assembly speedup macros. Only applies if `WOLFSSL_SP_MATH_ALL` is defined. See `sp_int.c`.

2.6.8.6.25 WOLFSSL_SP_RISCV64 Enable Single-Precision (SP) RISCV64 assembly speedup macros. Only applies if `WOLFSSL_SP_MATH_ALL` is defined. See `sp_int.c`.

2.6.8.6.26 WOLFSSL_SP_RISCV32 Enable Single-Precision (SP) RISCV32 assembly speedup macros. Only applies if `WOLFSSL_SP_MATH_ALL` is defined. See `sp_int.c`.

2.6.8.6.27 WOLFSSL_SP_S390X Enable Single-Precision (SP) S390X assembly speedup macros. Only applies if `WOLFSSL_SP_MATH_ALL` is defined. See `sp_int.c`.

2.6.8.7 SP_INT_BITS The number of bits to have in an sp_int. Which can determine the largest bignum that the library can handle.

2.6.9 Stack or Chip Specific Defines

wolfSSL can be built for a variety of platforms and TCP/IP stacks. Most of the following defines are located in `./wolfssl/wolfcrypt/settings.h` and are commented out by default. Each can be uncommented to enable support for the specific chip or stack referenced below.

2.6.9.1 IPHONE Can be defined if building for use with iOS.

2.6.9.2 THREADX Can be defined when building for use with the ThreadX RTOS (https://www.rtos.com).

2.6.9.3 MICRIUM Can be defined when building wolfSSL to enable support for Micrium's µC/OS-III RTOS (https://www.micrium.com).

2.6.9.4 MBED Can be defined when building for the mbed prototyping platform (https://www.mbed.org).

2.6.9.5 MICROCHIP_PIC32 Can be defined when building for Microchip's PIC32 platform (https://www.microchip.com).

2.6.9.6 MICROCHIP_TCPIP_V5 Can be defined specifically version 5 of microchip tcp/ip stack.

2.6.9.7 MICROCHIP_TCPIP Can be defined for microchip tcp/ip stack version 6 or later.

2.6.9.8 WOLFSSL_MICROCHIP_PIC32MZ Can be defined for PIC32MZ hardware cryptography engine.

2.6.9.9 FREERTOS Can be defined when building for FreeRTOS (https://www.freertos.org). If using LwIP, define `WOLFSSL_LWIP` as well.

2.6.9.10 FREERTOS_WINSIM Can be defined when building for the FreeRTOS windows simulator (https://www.freertos.org).

2.6.9.11 WOLFSSL_CHIBIOS Can be defined when building for ChibiOS RTOS.

2.6.9.12 WOLFSSL_CLEANUP_THREADSAFE_BY_ATOMIC_OPS Uses atomic operations to make `wolfSSL_Cleanup()` thread-safe without requiring a mutex.

2.6.9.13 WOLFSSL_CMSIS_RTOS Can be defined when building for Mbed CMIS-RTOS.

2.6.9.14 WOLFSSL_CMSIS_RTOSv2 Can be defined when building for Mbed CMIS-RTOSv2.

2.6.9.15 WOLFSSL_LWIP_NATIVE Use in platforms for LWIP native.

2.6.9.16 WOLFSSL_DEOS You can enable the wolfSSL support for Deos RTOS available here using this define.

2.6.9.17 WOLFSSL_ESPIDF Can be defined when building for ESP-IDF.

2.6.9.18 WOLFSSL_LINUXKM Use if building for Linux Kernel Module.

2.6.9.19 WORD64_AVAILABLE Portability macro to indicate 64-bit types are supported. Typically its better to use `SIZEOF_LONG_LONG` 8.

2.6.9.20 WOLFSSL_NUCLEUS_1_2 Use if building for Nucleus 1.2.

2.6.9.21 WOLFSSL_PICOTCP use if if building with PicoTCP.

2.6.9.22 WOLFSSL_RENESAS_RA6M3G Use if building with RENESAS RA6M3G.

2.6.9.23 WOLFSSL_RENESAS_RA6M4 use if building with RENESAS RA6M4.

2.6.9.24 WOLFSSL_RIOT_OS Use if building with RIOT-OS.

2.6.9.25 WOLFSSL_uITRON4 Use if building for uITRON4.

2.6.9.26 WOLFSSL_uTKERNEL2 Use if building with uT-Kernel.

2.6.9.27 WOLFSSL_VXWORKS Use if building with VxWorks.

2.6.9.28 DEVKITPRO Use when building for devkitPro.

2.6.9.29 WOLFSSL_VXWORKS_6_x Used only with implementation for VxWorks 6.x only.

2.6.9.30 WOLFSSL_VERIFY_CB_ALL_CERTS Calls the certificate verify callback for all certificates in the chain, not just the peer certificate.

2.6.9.31 WOLFSSL_WICED Used if building for WICED Studio.

2.6.9.32 FREESCALE_KSDK_FREERTOS Older name of this is FREESCALE_FREE_RTOS but this is used when building for Freescale KSDK FreeRTOS.

2.6.9.33 FREESCALE_KSDK_MQX Used when building for Freescale KSDK MQX/RTCS/MFS.

2.6.9.34 FREESCALE_MQX_5_0 Used when building for Freescale Classic MQX version 5.0.

2.6.9.35 WOLFSSL_KEIL_TCP_NET Configure the TCP stack (MDK_CONF_NETWORK). By default uses Keil TCP WOLFSSL_KEIL_TCP_NET. Use 0 for none or 2 for user io callbacks.

2.6.9.36 INTEL_GALILEO Used when configuring ARDUINO and wolfSSL. If building for Intel Galileo platform add: #define INTEL_GALILEO

2.6.9.37 HAVE_KEIL_RTX WolfSSL for MDK-RTX-TCP-FS Configuration.

2.6.9.38 EBSNET Can be defined when using EBSnet products and RTIP.

2.6.9.39 WOLFSSL_EMBOS Can be defined when building for SEGGER embOS (https://www.segger.com/products/rtos/embos/). If using emNET, define WOLFSSL_EMNET as well.

2.6.9.40 WOLFSSL_EMNET Can be defined when building for SEGGER emNET TCP/IP stack (https://www.segger.com/products/connectivity/emnet/).

2.6.9.41 WOLFSSL_LWIP Can be defined when using wolfSSL with the LwIP TCP/IP stack (https://savannah.nongnu.org/projects/lwip/).

2.6.9.42 WOLFSSL_ISOTP Can be defined when using wolfSSL with the ISO-TP transport protocol, typically used for CAN bus. A usage example can be found in the wolfssl-examples repository.

2.6.9.43 WOLFSSL_IOTSAFE Enables IoTSAFE (GSMA) applet support for secure element operations. Allows TLS keys and crypto to be handled by an IoTSAFE-compliant SIM.

2.6.9.44 WOLFSSL_GAME_BUILD Can be defined when building wolfSSL for a game console.

2.6.9.45 WOLFSSL_LSR Can be defined if building for LSR.

2.6.9.46 WOLFSSL_LOCAL_X509_STORE Uses a local X509 certificate store per SSL context instead of a global one. Provides better isolation between contexts.

2.6.9.47 FREESCALE_MQX Can be defined when building for Freescale MQX/RTCS/MFS (https://www.freescale.com). This in turn defines FREESCALE_K70_RNGA to enable support for the Kinetis H/W Random Number Generator Accelerator

2.6.9.48 WOLFSSL_STM32F2 Can be defined when building for STM32F2. This define also enables STM32F2 hardware crypto and hardware RNG support in wolfSSL (https://www.st.com/internet/mcu/subclass/1520.jsp).

2.6.9.49 COMVERGE Can be defined if using Comverge settings.

2.6.9.50 WOLFSSL_QL Can be defined if using QL SEP settings.

2.6.9.51 WOLFSSL_EROAD Can be defined building for EROAD.

2.6.9.52 WOLFSSL_IAR_ARM Can be defined if build for IAR EWARM.

2.6.9.53 WOLFSSL_TIRTOS Can be defined when building for TI-RTOS.

2.6.9.54 WOLFSSL_ROWLEY_ARM Can be defined when building with Rowley CrossWorks.

2.6.9.55 WOLFSSL_NRF51 Can be defined when porting to Nordic nRF51.

2.6.9.56 WOLFSSL_NRF51_AES Can be defined to use built-in AES hardware for AES 128 ECB encrypt when porting to Nordic nRF51.

2.6.9.57 WOLFSSL_CONTIKI Can be defined to enable support for the Contiki operating system.

2.6.9.58 WOLFSSL_COPY_CERT Copies the certificate buffer when loading into an SSL object instead of referencing it, ensuring the SSL object owns its own copy of the data.

2.6.9.59 WOLFSSL_COPY_KEY Copies the private key buffer when loading into an SSL object instead of referencing it, ensuring the SSL object owns its own copy of the data.

2.6.9.60 WOLFSSL_APACHE_MYNEWT Can be defined to enable the Apache Mynewt port layer.

2.6.9.61 WOLFSSL_APACHE_HTTPD Can be defined to enable support for the Apache HTTPD web server.

2.6.9.62 ASIO_USE_WOLFSSL Can be defined to make wolfSSL build as an ASIO-compatible version. ASIO then relies on the `BOOST_ASIO_USE_WOLFSSL` preprocessor define.

2.6.9.63 WOLFSSL_CRYPTOCELL Can be defined to enable using ARM CRYPTOCELL.

2.6.9.64 WOLFSSL_SIFIVE_RISC_V Can be defined to enable using RISC-V SiFive/HiFive port.

2.6.9.65 WOLFSSL_MDK_ARM Adds support for MDK ARM

2.6.9.66 WOLFSSL_MDK5 Adds support for MDK5 ARM

2.6.10 OS Specific Defines

2.6.10.1 USE_WINDOWS_API Specify use of windows library APIs' as opposed to Unix/Linux APIs'

2.6.10.2 WIN32_LEAN_AND_MEAN Adds support for the Microsoft win32 lean and mean build.

2.6.10.3 FREERTOS_TCP

Adds support for the FREERTOS TCP stack

2.6.10.4 WOLFSSL_SAFERTOS

Adds support for SafeRTOS

2.7 Build Options

The following are options which may be appended to the `./configure` script to customize how the wolfSSL library is built.

By default, wolfSSL only builds in shared mode, with static mode being disabled. This speeds up build times by a factor of two. Either mode can be explicitly disabled or enabled if desired.

2.7.1 --enable-debug

Enable wolfSSL debugging support. Enabling debug support allows easier debugging by compiling with debug information and defining the constant `DEBUG_WOLFSSL` which outputs messages to `stderr`. To turn debug on at runtime, call `wolfSSL_Debugging_ON()`. To turn debug logging off at runtime, call `wolfSSL_Debugging_OFF()`. For more information, see Debugging.

2.7.2 --enable-distro

Enable wolfSSL distro build.

2.7.3 --enable-singlethread

Enable single threaded mode, no multi thread protections.

Enabling single threaded mode turns off multi thread protection of the session cache. Only enable single threaded mode if you know your application is single threaded or your application is multithreaded and only one thread at a time will be accessing the library.

2.7.4 --enable-dtls

Enable wolfSSL DTLS support

Enabling DTLS support allows users of the library to also use the DTLS protocol in addition to TLS and SSL. For more information, see the DTLS section.

2.7.5 --disable-rng

Disable compiling and using RNG

2.7.6 --enable-sctp

Enable wolfSSL DTLS-SCTP support

2.7.7 --enable-openssh

Enable OpenSSH compatibility build

2.7.8 --enable-apachehttpd

Enable Apache httpd compatibility build

2.7.9 --enable-openvpn

Enable OpenVPN compatibility build

2.7.10 --enable-opensslextra

Enable extra OpenSSL API compatibility, increases the size

Enabling OpenSSL Extra includes a larger set of OpenSSL compatibility functions. The basic build will enable enough functions for most TLS/SSL needs, but if you're porting an application that uses 10s or 100s of OpenSSL calls, enabling this will allow better support. The wolfSSL OpenSSL compatibility layer is under active development, so if there is a function missing which you need, please contact us and we'll try to help. For more information about the OpenSSL Compatibility Layer, please see OpenSSL Compatibility.

2.7.11 --enable-opensslall

Enable all OpenSSL API.

2.7.12 --enable-maxstrength

Enable Max Strength build, allows TSLv1.2-AEAD-PFS ciphers only. This is disabled by default because it can cause interoperability issues. It also enables glitching detection.

2.7.13 --disable-harden

Disable hardening, timing resistance and RSA blinding. Disabling this feature can give performance improvements.

NOTE Hardening provides mitigations against side channel attacks. Only disable this feature after careful consideration.

To disable via user_settings.h the equivalent settings would be:

- `#define WC_NO_CACHE_RESISTANT`
- `#define WC_NO_HARDEN`
- Remove the setting `WC_RSA_BLINDING` if present or un-define it
- Remove the setting `ECC_TIMING_RESISTANT` if present or un-define it
- Remove the setting `TFM_TIMING_RESISTANT` if present or un-define it

2.7.14 --enable-ipv6

Enable testing of IPv6, wolfSSL proper is IP neutral

Enabling IPV6 changes the test applications to use IPv6 instead of IPv4. wolfSSL proper is IP neutral, either version can be used, but currently the test applications are IP dependent.

2.7.15 --enable-bump

Enable SSL Bump build

2.7.16 --enable-leanpsk

Enable Lean PSK build.

Very small build using PSK, and eliminating many features from the library. Approximate build size for wolfSSL on an embedded system with this enabled is 21kB.

2.7.17 --enable-leantls

Implements a lean TLS 1.2 client only (no client auth), ECC256, AES128 and SHA256 w/o Shamir. Meant to be used by itself at the moment and not in conjunction with other build options.

Enabling produces a small footprint TLS client that supports TLS 1.2 client only (no client auth), ECC256, AES128 and SHA256 w/o Shamir. Meant to be used by itself at the moment and not in conjunction with other build options.

2.7.18 --enable-tinytls13

Enable the tiny TLS 1.3 footprint profile. This is a TLS 1.3 only build that strips wolfSSL to a minimal handshake surface for embedded and resource constrained targets. It is expressed as a translation layer over standard wolfSSL macros (the `WOLFSSL_TINY_TLS13` umbrella in `settings.h`), so it adds no new feature flags to the core library.

The profile is selected with a comma separated list:

```
./configure --enable-tinytls13=LIST
```

The bare flag `--enable-tinytls13` is equivalent to `--enable-tinytls13=psk`. This profile is non FIPS and cannot be combined with a FIPS build.

2.7.18.1 Base profiles

- `psk` (default): a pre shared key plus ECDHE build with no X.509. The floor is X25519 key exchange, AES-128-GCM, SHA-256 and HKDF. This is the smallest configuration.
- `cert`: adds a minimal X.509 certificate chain verify on top of the floor, using ECDSA P-256 by default.

The certificate profile is a reduced security verify. It disables X.509 name constraint enforcement, relaxes ASN.1 strictness and removes CRL revocation checking to save code. It is intended for a known or pinned certificate authority, not for general public internet PKI. Configure prints a notice when the certificate profile is selected.

2.7.18.2 Adders The following may be combined with a base profile in the comma separated list:

- `server`: add the TLS 1.3 server role. The default is client only.
- `mutualauth`: mutual TLS with X.509 client authentication. Implies `cert`.
- `staticmem`: serve TLS allocations from a caller provided static memory pool. Add `WOLFSSL_NO_MALLOC` separately for a true zero heap build.
- `asm`: use assembly crypto instead of the small C backend. Larger but faster.
- p256: use P-256 ECDHE on the PSK floor instead of X25519.
- sha384: add SHA-384, for example for AES-256-GCM-SHA384.
- `mldsa`: add ML-DSA-65 certificate verification (verify only).
- `rsaverify`: add RSA-PSS certificate verification. The certificate profile is ECDSA only by default.

For example, a TLS 1.3 server that verifies ECDSA certificates and also supports SHA-384:

```
./configure --enable-tinytls13=cert,server,sha384
```

2.7.18.3 Footprint The figures below are flash (text plus data) of the linked client binary, built `-Os` with link time optimization and dead code elimination (`-flto -ffunction-sections -fdata-sections -Wl,--gc-sections`) on the SHA-256 only floor. This is what links into an application, which is much smaller than the size of `libwolfssl.a`. Toolchains: arm-none-eabi-gcc 14.2 with newlib-nano (Cortex-M33), clang `-Os` (Intel x86_64), aarch64-none-elf-gcc 14.2 with newlib (ARM aarch64).

Configuration	Cortex-M33	Intel x86_64	ARM aarch64
PSK, X25519 (floor)	30,836	56,092	50,700
PSK, P-256	37,347	66,068	59,292
Cert, ECDSA P-256	62,765	110,817	95,404
Mutual TLS	66,554	116,789	99,836

Cost of common adders on Cortex-M33, relative to the PSK floor:

Build	Flash (bytes)	Delta vs floor
PSK floor (X25519)	30,836	0
plus static memory (zero heap)	32,169	+1,333
plus P-256	37,347	+6,511
plus ML-DSA-65 verify	43,780	+12,944

Where the flash goes for a PSK P-256 client on Cortex-M33 (37,347 bytes):

Component	Bytes
TLS 1.3 handshake	7,766
TLS extensions	5,678
ECC P-256 (SP math)	5,962
CTX, SSL object and API	4,050
Record layer	3,564
AES-128-GCM	2,650
HMAC, HKDF, DRBG and RNG	2,110
SHA-256	1,370
Key schedule	1,120
Memory and cleanup	630
C library (newlib)	952
Other and inlined	1,486

2.7.18.4 Customizing for your application

1. Start from the closest base profile. Use `psk` if your endpoints share a pre shared key and you do not need certificates. Use `cert` if you must validate an X.509 certificate from a known certificate authority.
2. Add only the adders you need. Each one is a deliberate, measured step up in size, as the tables above show. Begin with `server` if the build is a server, and add a curve or signature adder only when your peer requires it.
3. Choose the key exchange curve. The PSK floor uses X25519 and the certificate profile uses P-256. Add p256 when you want P-256 on the PSK floor, for example to reuse the P-256 code already pulled in by an ECDSA certificate. To supply your own group instead, for example P-384 or a post quantum hybrid, select that curve yourself and define `WOLFSSL_TINY_TLS13_NO_DEFAULT_CURVE` so the profile does not force a default X25519 or P-256.
4. Choose the memory model. The default uses the system allocator. Add `staticmem` to serve TLS allocations from a fixed pool supplied with `wolfSSL_CTX_load_static_memory()`. For a target with no heap at all, also define `WOLFSSL_NO_MALLOC`.
5. Provide the platform glue for bare metal. The example template enables `WOLFSSL_USER_IO` (you supply send and receive callbacks), `NO_FILESYSTEM` and `WOLFSSL_NO_SOCK`, and expects a hardware entropy source through `CUSTOM_RAND_GENERATE_SEED`.

6. Build for size. Link your application with `-Os -flto -ffunction-sections -fdata-sections -Wl,--gc-sections` so unused code is dropped.

The same profile is available without autotools through the `WOLFSSL_TINY_TLS13` umbrella in `settings.h`. The template `examples/configs/user_settings_tinytls13.h` selects the profile and adders with `#if 0` and `#if 1` blocks. The smallest result comes from a dead code eliminated, link time optimized build:

```
cp ./examples/configs/user_settings_tinytls13.h user_settings.h
./configure --enable-usersettings --enable-static --disable-shared \
            --disable-examples --disable-crypttests
make
# link your application with:
#   -Os -flto -ffunction-sections -fdata-sections -Wl,--gc-sections
```

The profile is driven entirely by the `WOLFSSL_TINY_TLS13` family of macros. Each one maps to a configure list item, so a `settings.h` build and an `--enable-tinytls13` build select the same features:

Macro	Configure item	Effect
`WOLFSSL_TINY_TLS13`	`psk`	The base profile. TLS 1.3 only, PSK plus ECDHE, no X.509. X25519, AES-128-GCM, SHA-256 and HKDF. Every other macro implies this one.
`WOLFSSL_TINY_TLS13_CERT`	`cert`	Add minimal X.509 certificate chain verify (ECDSA P-256), and switch the default curve to P-256. Implies the base profile.
`WOLFSSL_TINY_TLS13_MUTUAL_AUTH`	`mutualauth`	Mutual TLS with X.509 client authentication. Implies `WOLFSSL_TINY_TLS13_CERT`.
`WOLFSSL_TINY_TLS13_SERVER`	`server`	Add the TLS 1.3 server role. The default is client only.
`WOLFSSL_TINY_TLS13_STATIC_MEM`	`staticmem`	Serve TLS allocations from a caller provided static memory pool.
`WOLFSSL_TINY_TLS13_ASM`	`asm`	Use assembly crypto instead of the small C backend. Larger but faster.
`WOLFSSL_TINY_TLS13_RSA_VERIFY`	`rsaverify`	Add RSA-PSS certificate verification. Pair with the certificate profile.
`WOLFSSL_TINY_TLS13_NO_DEFAULT_CURVE`	(none)	Suppress the automatic default curve. The profile then enables neither X25519 nor P-256, so you can select your own group (for example P-384 or a post quantum hybrid).

The `p256`, `sha384` and `mldsa` configure items have no dedicated tiny macro; they are selected with the standard wolfSSL macros (`HAVE_ECC` with `ECC_USER_CURVES`, `WOLFSSL_SHA384`, and `WOLFSSL_HAVE_MLDSA` with `WOLFSSL_MLDSA_VERIFY_ONLY`). See `examples/configs/user_settings_tinytls13.h` for the exact blocks.

A self contained, single process TLS 1.3 handshake smoke test for the profile is provided in `examples/configs/tinytls13_smoke.c`. It wires an in memory client and server together with no sockets or threads, so it can validate a tiny build that has no example or test harness available.

2.7.18.5 Dos and don'ts

- Do use the `cert` profile only with a known or pinned certificate authority. Its verify is deliberately reduced (no name constraints, relaxed ASN.1, no CRL).
- Do not use the `cert` profile for general public internet PKI.
- Do pair `mldsa` and `rsaverify` with the `cert` profile. They are certificate verification algorithms, so on the PSK floor they only confirm the build links.
- Do supply a real hardware entropy source through `CUSTOM_RAND_GENERATE_SEED` on bare metal. The default seed sources are not present once `NO_FILESYSTEM` and similar are set.
- Do not combine the profile with a FIPS build. It is a non FIPS footprint profile.
- Do not define `WOLFSSL_NO_MALLOC` and expect the standard `make check` test suite to run, since that suite needs an allocator. Validate a zero heap build by linking it and running the smoke test instead.
- Note that AES-256-GCM-SHA384 uses a SHA-384 transcript. With an external pre shared key, provision the key for SHA-384 through the cipher suite specific PSK callback, or use the cipher on the certificate profile, otherwise the PSK binder will not match.

2.7.19 `--enable-bigcache`

Enable a big session cache.

Enabling the big session cache will increase the session cache from 33 sessions to 20,027 sessions. The default session cache size of 33 is adequate for TLS clients and embedded servers. The big session cache is suitable for servers that aren't under heavy load, basically allowing 200 new sessions per minute or so.

2.7.20 `--enable-hugecache`

Enable a huge session cache.

Enabling the huge session cache will increase the session cache size to 65,791 sessions. This option is for servers that are under heavy load, over 13,000 new sessions per minute are possible or over 200 new sessions per second.

2.7.21 `--enable-smallcache`

Enable small session cache.

Enabling the small session cache will cause wolfSSL to only store 6 sessions. This may be useful for embedded clients or systems where the default of nearly 3kB is too much RAM. This define uses less than 500 bytes of RAM.

2.7.22 `--enable-savesession`

Enable persistent session cache.

Enabling this option will allow an application to persist (save) and restore the wolfSSL session cache to/from memory buffers.

2.7.23 --enable-savecert

Enable persistent cert cache.

Enabling this option will allow an application to persist (save) and restore the wolfSSL certificate cache to/from memory buffers.

2.7.24 --enable-atomicuser

Enable Atomic User Record Layer.

Enabling this option will turn on User Atomic Record Layer Processing callbacks. This will allow the application to register its own MAC/encrypt and decrypt/verify callbacks.

2.7.25 --enable-pkcallbacks

Enable Public Key Callbacks

Enabling this option will turn on Public Key callbacks, allowing the application to register its own ECC sign/verify and RSA sign/verify and encrypt/decrypt callbacks.

2.7.26 --enable-sniffer

Enable wolfSSL sniffer support.

Enabling sniffer (SSL inspection) support will allow the collection of SSL traffic packets as well as the ability to decrypt those packets with the correct key file.

Currently the sniffer supports the following RSA ciphers:

CBC ciphers:

- AES-CBC
- Camellia-CBC
- 3DES-CBC

Stream ciphers:

- RC4

2.7.27 --enable-aesgcm

Enable AES-GCM support.

Enabling AES-GCM will add these cipher suites to wolfSSL. wolfSSL offers four different implementations of AES-GCM balancing speed versus memory consumption. If available, wolfSSL will use 64-bit or 32-bit math. For embedded applications, there is a speedy 8-bit version that uses RAM-based lookup tables (8KB per session) which is speed comparable to the 64-bit version and a slower 8-bit version that doesn't take up any additional RAM. The –enable-aesgcm configure option may be modified with the options `=word32`, `=table`, or `=small`, i.e. `--enable-aesgcm=table`.

2.7.28 --enable-aesccm

Enable AES-CCM support

2.7.29 --disable-aescbc

Used to with `--disable-aescbc` to compile out AES-CBC

AES-GCM will enable Counter with CBC-MAC Mode with 8-byte authentication (CCM-8) for AES.

2.7.30 --enable-aescfb

Turns on AES-CFB mode support

2.7.31 --enable-aesctr

Enable wolfSSL AES-CTR support

Enabling AES-CTR will enable Counter mode.

2.7.32 --enable-aesni

Enable wolfSSL Intel AES-NI support

Enabling AES-NI support will allow AES instructions to be called directly from the chip when using an AES-NI supported chip. This provides speed increases for AES functions. See Features for more details regarding AES-NI.

2.7.33 --enable-intelasm

Enable ASM speedups for Intel and AMD processors.

Enabling the intelasm option for wolfSSL will utilize expanded capabilities of your processor that dramatically enhance AES performance. The instruction sets leveraged when configure option is enabled include AVX1, AVX2, BMI2, RDRAND, RDSEED, AESNI, and ADX. These were first introduced into Intel processors and AMD processors have started adopting them in recent years. When enabled, wolfSSL will check the processor and take advantage of the instruction sets your processor supports.

2.7.34 --enable-camellia

Enable Camellia support

2.7.35 --enable-md2

Enable MD2 support

2.7.36 --enable-nullcipher

Enable wolfSSL NULL cipher support (no encryption)

2.7.37 --enable-ripemd

Enable wolfSSL RIPEMD-160 support

2.7.38 --enable-blake2

Enable wolfSSL BLAKE2 support

2.7.39 --enable-blake2s

Enable wolfSSL BLAKE2s support

2.7.40 --enable-sha3

Enabled by default on x86_64 and Aarch64.

Enables wolfSSL SHA3 support (=small for small build)

2.7.41 --enable-sha512

Enabled by default on x86_64.

Enable wolfSSL SHA-512 support

2.7.42 --enable-she

Enable SHE (Secure Hardware Extension) key update message generation support. SHE is an automotive security standard for secure key management in ECUs. This module provides software-based generation and verification of SHE key update messages (M1-M5) with optional hardware offload via crypto callbacks.

Two modes are available:

- `--enable-she=standard` - Standard SHE support (defines `WOLFSSL_SHE`)
- `--enable-she=extended` - Standard plus extended overrides for custom KDF constants and message headers (defines `WOLFSSL_SHE` and `WOLFSSL_SHE_EXTENDED`)

SHE automatically enables its dependencies: AES, AES-CBC, AES-direct, and CMAC.

The following defines can be used to compile out optional features:

- `NO_WC_SHE_GETUID` - Compile out `wc_SHE_GetUID` callback
- `NO_WC_SHE_GETCOUNTER` - Compile out `wc_SHE_GetCounter` callback
- `NO_WC_SHE_IMPORT_M123` - Compile out `wc_SHE_ImportM1M2M3`
- `NO_WC_SHE_EXPORTKEY` - Compile out `wc_SHE_ExportKey` callback
- `NO_WC_SHE_LOADKEY` - Compile out all `wc_SHE_LoadKey` / `wc_SHE_LoadKey_Verify` convenience wrappers
- `WC_SHE_SW_DEFAULT` - Enable software-only default UID and counter for testing without hardware

2.7.43 --enable-sessioncerts

Enable session cert storing

2.7.44 --enable-keygen

Enable key generation (applies to RSA key generation only)

2.7.45 --enable-certgen

Enable cert generation

2.7.46 --enable-certext

Enable cert extensions (see chapter 7 on supported extensions)

2.7.47 --enable-certreq

Enable cert request generation

2.7.48 --enable-sep

Enable SEP extensions

2.7.49 --enable-hkdf

Enable HKDF (HMAC-KDF)

2.7.50 --enable-x963kdf

Enable X9.63 KDF support

2.7.51 --enable-dsa

Enable Digital Signature Algorithm (DSA).

NIST approved digital signature algorithm along with RSA and ECDSA as defined by FIPS 186-4 and are used to generate and verify digital signatures if used in conjunction with an approved hash function as defined by the Secure Hash Standard (FIPS 180-4).

2.7.52 --enable-eccshamir

Enabled by default on x86_64

Enable ECC Shamir

2.7.53 --enable-ecc

Enabled by default on x86_64

Enable ECC.

Enabling this option will build ECC support and cipher suites into wolfSSL.

2.7.54 --enable-ecccustcurves

Enable ECC custom curves (`=all` to enable all curve types)

2.7.55 --enable-compkey

Enable compressed keys support

2.7.56 --enable-curve25519

Enable Curve25519 (or `--enable-curve25519=small` for CURVE25519_SMALL).

An elliptic curve offering 128 bits of security and to be used with ECDH key agreement (see Cross Compiling). Enabling curve25519 option allows for the use of the curve25519 algorithm. The default curve25519 is set to use more memory but have a faster run time. To have the algorithm use less memory the option `--enable-curve25519=small` can be used. Although using less memory there is a trade off in speed.

2.7.57 --enable-ed25519

Enable ED25519 (or `--enable-ed25519=small` for ED25519_SMALL)

Enabling ed25519 option allows for the use of the ed25519 algorithm. The default ed25519 is set to use more memory but have a faster run time. To have the algorithm use less memory the option `--enable-ed25519=small` can be used. Like with curve25519 using this enable option less is a trade off between speed and memory.

2.7.58 --enable-fpecc

Enable Fixed Point cache ECC

2.7.59 --enable-eccencrypt

Enable ECC encrypt

2.7.60 --enable-psk

Enable PSK (Pre Shared Keys)

2.7.61 --disable-errorstrings

Disable the error strings table

2.7.62 --disable-oldtls

Disable old TLS version < 1.2. This is the default; the corresponding `--enable-oldtls` leaves `NO_OLD_TLS` undefined.

Warning: Building with `--enable-oldtls` is **not conformant** with RFC 9846, RFC 8996 or RFC 9325. RFC 8996 deprecates TLS 1.0 and TLS 1.1, and RFC 9846 states that SSL 2.0, SSL 3.0, TLS 1.0 and TLS 1.1 "MUST NOT be negotiated for any reason".

2.7.63 --enable-tlsv10

Enable TLS version 1.0 (defines `WOLFSSL_ALLOW_TLSV10`, and enables old TLS).

Warning: Building with `--enable-tlsv10` is **not conformant** with RFC 9846, RFC 8996 or RFC 9325. See `--disable-oldtls` above.

2.7.64 --enable-sslv3

Enable SSL version 3.0 (defines `WOLFSSL_ALLOW_SSLV3`, and enables old TLS).

Warning: Building with `--enable-sslv3` violates RFC 7568. SSL 3.0 is deprecated by RFC 7568 and states "SSLv3 MUST NOT be used."

2.7.65 --enable-stacksize

Enable stack size info on examples

2.7.66 --disable-memory

Disable memory callbacks

2.7.67 --disable-rsa

Disable RSA

2.7.68 --enable-rsapss

Enable RSA-PSS

2.7.69 --disable-dh

Disable DH

2.7.70 --enable-anon

Enable Anonymous

2.7.71 --disable-asn

Disable ASN

2.7.72 --disable-aes

Disable AES

2.7.73 --disable-coding

Disable Coding base 16/64

2.7.74 --enable-base64encode

Enabled by default on x86_64
Enable Base64 encoding

2.7.75 --disable-des3

Disable DES3

2.7.76 --enable-arc4

Enable ARC4

2.7.77 --disable-md5

Disable MD5

2.7.78 --disable-sha

Disable SHA

2.7.79 --enable-webserver

Enable Web Server.

This turns on functions required over the standard build that will allow full functionality for building with the yaSSL Embedded Web Server.

2.7.80 --enable-fips

Enable FIPS 140-2 (Must have license to implement.)

2.7.81 --enable-sha224

Enabled by default on x86_64

Enable wolfSSL SHA-224 support

2.7.82 --disable-poly1305

Disable wolfSSL POLY1305 support

2.7.83 --disable-chacha

Disable CHACHA

2.7.84 --disable-hashdrbg

Disable Hash DRBG support

2.7.85 --disable-filesystem

Disable Filesystem support.

This makes it easier to disable filesystem use. This option defines `NO_FILESYSTEM`.

2.7.86 --disable-inline

Disable inline functions.

Disabling this option disables function inlining in wolfSSL. Function placeholders that are not linked against but, rather, the code block is inserted into the function call when function inlining is enabled.

2.7.87 --enable-ocsp

Enable Online Certificate Status Protocol (OCSP).

Enabling this option adds OCSP (Online Certificate Status Protocol) support to wolfSSL. It is used to obtain the revocation status of x.509 certificates as described in RFC 6960.

2.7.88 --enable-ocspstapling

Enable OCSP Stapling (=no-multi to disable multiple OCSP Stapling for TLS1.3 Certificate)

2.7.89 --enable-ocspstapling2

Enable OCSP Stapling version 2

2.7.90 --enable-crl

Enable CRL (Certificate Revocation List)

2.7.91 --enable-crl-monitor

Enable CRL Monitor.

Enabling this option adds the ability to have wolfSSL actively monitor a specific CRL (Certificate Revocation List) directory.

2.7.92 --enable-sni

Enable Server Name Indication (SNI).

Enabling this option will turn on the TLS Server Name Indication (SNI) extension.

2.7.93 --enable-maxfragment

Enable Maximum Fragment Length.

Enabling this option will turn on the TLS Maximum Fragment Length extension.

2.7.94 --enable-alpn

Enable Application Layer Protocol Negotiation (ALPN)

2.7.95 --enable-truncatedhmac

Enable Truncated Keyed-hash MAC (HMAC).

Enabling this option will turn on the TLS Truncated HMAC extension.

2.7.96 --enable-renegotiation-indication

Enable Renegotiation Indication.

As described in RFC 5746, this specification prevents an SSL/TLS attack involving renegotiation splicing by tying the renegotiations to the TLS connection they are performed over.

2.7.97 --enable-secure-renegotiation

Enable Secure Renegotiation

2.7.98 --enable-supportedcurves

Enable Supported Elliptic Curves.

Enabling this option will turn on the TLS Supported ECC Curves extension.

2.7.99 --enable-session-ticket

Enable Session Ticket

2.7.100 --enable-extended-master

Enable Extended Master Secret

2.7.101 --enable-tlsx

Enable all TLS extensions.

Enabling this option will turn on all TLS extensions currently supported by wolfSSL.

2.7.102 --enable-pkcs7

Enable PKCS#7 support

2.7.103 --enable-pkcs11

Enable PKCS#11 access

2.7.104 --enable-ssh

Enable wolfSSH options

2.7.105 --enable-scep

Enable wolfSCEP (Simple Certificate Enrollment Protocol)

As defined by the Internet Engineering Task Force, Simple Certificate Enrollment Protocol is a PKI that leverages PKCS#7 and PKCS#10 over HTTP. CERT notes that SCEP does not strongly authenticate certificate requests.

2.7.106 --enable-srp

Enable Secure Remote Password

2.7.107 --enable-smallstack

Enable Small Stack Usage

2.7.108 --enable-valgrind

Enable valgrind for unit tests.

Enabling this option will turn on valgrind when running the wolfSSL unit tests. This can be useful for catching problems early on in the development cycle.

2.7.109 --enable-testcert

Enable Test Cert.

When this option is enabled, it exposes part of the ASN certificate API that is usually not exposed. This can be useful for testing purposes, as seen in the wolfCrypt test application (`wolfcrypt/test/test.c`).

2.7.110 --enable-iopool

Enable I/O Pool example

2.7.111 --enable-certservice

Enable certificate service (Windows Servers)

2.7.112 --enable-jni

Enable wolfSSL JNI

2.7.113 --enable-lighty

Enable lighttpd/lighty

2.7.114 --enable-stunnel

Enable stunnel

2.7.115 --enable-md4

Enable MD4

2.7.116 --enable-pwdbased

Enable PWDBASED

2.7.117 --enable-scrypt

Enable SCRYPT

2.7.118 --enable-cryptonly

Enable wolfCrypt Only build

2.7.119 --disable-examples

Disable building examples.

When enabled, the wolfSSL example applications will be built (client, server, echoclient, echoserver).

2.7.120 --disable-crypttests

Disable Crypt Bench/Test

2.7.121 --enable-fast-rsa

Enable RSA using Intel IPP.

Enabling fast-rsa speeds up RSA operations by using IPP libraries. It has a larger memory consumption then the default RSA set by wolfSSL. If IPP libraries can not be found an error message will be displayed during configuration. The first location that autoconf will look is in the directory `wolfssl_root/IPP` the second is standard location for libraries on the machine such as `/usr/lib/` on linux systems.

The libraries used for RSA operations are in the directory `wolfssl-X.X.X/IPP/` where X.X.X is the current wolfSSL version number. Building from the bundled libraries is dependent on the directory location and name of IPP so the file structure of the subdirectory IPP should not be changed.

When allocating memory the fast-rsa operations have a memory tag of `DYNAMIC_TYPE_USER_CRYPTO`. This allows for viewing the memory consumption of RSA operations during run time with the fast-rsa option.

2.7.122 --enable-staticmemory

Enable static memory use

2.7.123 --enable-mcapi

Enable Microchip API

2.7.124 --enable-asynccrypt

Enable Asynchronous Crypto

2.7.125 --enable-sessionexport

Enable export and import of sessions

2.7.126 --enable-aeskeywrap

Enable AES key wrap support

2.7.127 --enable-jobserver

Values: `yes` (default) / no / #

When using `make` this builds wolfSSL using a multithreaded build, `yes` (default) detects the number of CPU cores and builds using a recommended amount of jobs for that count, # to specify an exact number. This works in a similar way to the `make -j` option.

2.7.128 --enable-shared[=PKGS]

Building shared wolfSSL libraries [default=yes]

Disabling the shared library build will exclude a wolfSSL shared library from being built. By default only a shared library is built in order to save time and space.

2.7.129 --enable-static[=PKGS]

Building static wolfSSL libraries [default=no]

2.7.130 --with-liboqs=PATH

Path to OpenQuantumSafe install (default `/usr/local`).

This turns on the ability for wolfSSL to use the experimental TLS 1.3 quantum-safe KEM groups, hybrid quantum-safe KEM groups and FALCON signature scheme via wolfSSL integration with liboqs. Please see the appendix “Experimenting with Quantum-Safe Cryptography” in this document for more details.

2.7.131 --with-libz=PATH

Optionally include libz for compression.

Enabling libz will allow compression support in wolfSSL from the libz library. Think twice about including this option and using it by calling `wolfSSL_set_compression()`. While compressing data before sending decreases the actual size of the messages being sent and received, the amount of data saved by compression usually takes longer in time to analyze than it does to send it raw on all but the slowest of networks.

2.7.132 --with-cavium

Path to cavium/software directory.

2.7.133 --with-user-crypto

Path to USER_CRYPTO install (default `/usr/local`).

2.7.134 --enable-rsavfy

Enables RSA verify only support (**note** requires `--enable-cryptonly`)

2.7.135 --enable-rsapub

Enables RSA public key only support (**note** requires `--enable-cryptonly`)

2.7.136 --enable-armasm

Enables ARMv8 ASM support.

The default configure sets mcpu or mfpu based on 64 vs 32 bit system. It does not overwrite mcpu or mfpu setting passed in by use of CPPFLAGS. On some compilers `-mstrict-align` may be needed due to the constraiants and `-mstrict-align` is now also set by default unless a user passes in mcpu/mfpu flags with CPPFLAGS.

2.7.137 --disable-tlsv12

Disable TLS 1.2 support

2.7.138 --enable-tls13

Enable TLS 1.3 support

This build option can be combined with `--disable-tlsv12` and `--disable-oldtls` to produce a wolfSSL build that is only TLS 1.3.

2.7.139 --enable-all

Enables all wolfSSL features, excluding SSL v3

2.7.140 --enable-xts

Enables AES-XTS mode

2.7.141 --enable-asio

Enables ASIO.

Requires that the options `--enable-opensslextra` and `--enable-opensslall` be enabled when configuring wolfSSL. If these two options are not enabled, then the autoconf tool will automatically enable these options to enable ASIO when configuring wolfSSL.

2.7.142 --enable-qt

Enables Qt 5.12 onwards support.

Enables wolfSSL build settings compatible with the wolfSSL Qt port. Patch file is required from wolfSSL for patching Qt source files.

2.7.143 --enable-qt-test

Enable Qt test compatibility build.

Enables support for building wolfSSL for compatibility with running the built-in Qt tests.

2.7.144 --enable-apache-httpd

Enables Apache httpd support

2.7.145 --enable-afalg

Enables use of Linux module AF_ALG for hardware acceleration. Additional Xilinx use with `=xilinx`, `=xilinx-rsa`, `=xilinx-aes`, `=xilinx-sha3`

Is similar to `--enable-devcrypto` in that it leverages a Linux kernel module (AF_ALG) for offloading crypto operations. On some hardware the module has performance accelerations available through the Linux crypto drivers. In the case of Petalinux with Xilinx the flag `--enable-afalg=xilinx` can be used to tell wolfSSL to use the Xilinx interface for AF_ALG.

2.7.146 --enable-devcrypto

Enables use of Linux `/dev/crypto` for hardware acceleration.

Has the ability to receive arguments, being able to receive any combination of `aes` (all aes support), `hash` (all hash algorithms), and `cbc` (aes-cbc only). If no options are given, it will default to using `all`.

2.7.147 --enable-mcast

Enable wolfSSL DTLS multicast support

2.7.148 --disable-pkcs12

Disable PKCS12 code

2.7.149 --enable-fallback-scsv

Enables Signaling Cipher Suite Value(SCSV)

2.7.150 --enable-psk-one-id

Enables support for single PSK ID with TLS 1.3

2.7.151 --enable-cryptocb

Enable crypto callbacks. Register a crypto callback using wc_CryptoCb_RegisterDevice and set the associated devId using wolfSSL_CTX_SetDevId.

The following two defines can be used with `--enable-cryptocb` to compile out RSA or ECC software fallback to optimize for footprint reduction when software RSA/ECC is not required.

- WOLF_CRYPTO_CB_ONLY_RSA - compiles out RSA software crypto fallback
- WOLF_CRYPTO_CB_ONLY_ECC - compiles out ECC software crypto fallback

Use of the WOLF_CRYPTO_CB_ONLY_* options requires disabling the examples. See `--disable-examples`

2.7.152 --enable-reproducible-build

Suppresses the binary jitter (timestamps and other non-functional metadata) to allow generation of bitwise-identical binary packages with identical hashes.

2.7.153 --enable-sys-ca-certs

Allows wolfSSL to use trusted system CA certificates for verification when `wolfSSL_CTX_load_system_CA_certs()` is called, either by loading them into wolfSSL certificate manager, or by invoking system authentication APIs. See `wolfSSL_CTX_load_system_CA_certs()` for more details.

2.8 Special Math Optimization Flags

2.8.1 --enable-fastmath

Enable FastMath implementation. Both FastMath and Big Integer library are disabled if Single-Precision (SP) math is enabled.

See USE_FAST_MATH and Big Integer Math Library sections.

2.8.2 --enable-fasthugemath

Enable fast math + huge code.

Enabling fasthugemath includes support for the FastMath library and greatly increases the code size by unrolling loops for popular key sizes during public key operations. Try using the benchmark utility before and after using fasthugemath to see if the slight speedup is worth the increased code size.

2.8.3 --enable-sp-math

Enable Single-Precision (SP) math implementation with restricted algorithm suite. Unsupported algorithms are disabled. Overrides `--enable-sp`, `--enable-sp-math-all`, `--enable-fastmath` and `--enable-fasthugemath`.

- Replaces the math implementation with that in sp_int.c
- A minimal implementation, turns on portions of sp_int.c but not all.
- MUST combine with –enable-sp to turn on the solutions in sp_x86_64.c or sp_arm.c etc (list of files below depending on the target system) to be able to perform RSA/ECC/DH operations.
- Not to be combined with –enable-sp-math-all (below)

FILE LIST (platform dependent, chosen by configure based on system specs or can be manually controlled when using a Makefile/IDE solution): sp_arm32.c sp_arm64.c sp_armthumb.c sp_cortexm.c sp_dsp32.c sp_x86_64.c sp_x86_64_asm.S sp_x86_64_asm.asm

2.8.4 --enable-sp-math-all

Enabled by default. Enable Single-Precision (SP) math implementation with full algorithm suite. Unsupported algorithms are enabled, but unoptimized. Overrides `--enable-sp`, `--enable-fastmath` and `--enable-fasthugemath`.

- Replaces the math implementation with that in sp_int.c
- A FULL implementation, does not depend on –enable-sp to work
- Can be combined with –enable-sp to allow use of the implementations written in portable c assembly (non-hardware specific assembly) in sp_c32.c for 32-bit or sp_c64.c for 64-bit when possible. The rest of the time (when not possible) the implementations in sp_int.c are used. The portable C assembly gives significant performance gains on targets that do not have hardware optimizations available.
- Not to be combined with –enable-sp-math (above)

NOTE: If you are using asymmetric cryptography with key length in bits [256, 384, 521, 1024, 2048, 3072, 4096], you should consider using –enable-sp-math option to get maximum performance with a larger footprint size.

2.8.5 --enable-sp-asm

Enable Single-Precision (SP) assembly implementation.

Can be used to enable Single-Precision performance improvements through assembly with Intel x86_64 and ARM architectures.

2.8.6 --enable-sp=OPT

Enable Single-Precision (SP) math for RSA, DH, and ECC to improve performance.

There are many possible values for OPT. Below is a list of ways to call enable-sp and the resulting macros that will be defined as a result. All of these can be combined in a coma separated list. For example, `--enable-sp=ec256,ec384`. The meaning of the macros that will be defined are defined above in the [wolfSSL's Proprietary Single Precision (SP) Math Support] section.

NOTE: 1) "–enable-sp=small –enable-sp-math" can be smaller than... 2) "–enable-sp-math-all=small"... as (1) only has implementations of specific key sizes while (2) has implementations to support all key sizes.

NOTE: This is for x86_64 and with no other configuration flags; your results may vary depending on your architectures and other configuration flags that you specify. For example, WOLFSSL_SP_384, WOLFSSL_SP_521 and WOLFSSL_SP_4096 will only be enabled for 64-bit Intel/ARM targets (x86_64, amd64, aarch64).

2.8.6.1 --enable-sp=no or --disable-sp

No new macros defined. Equivalent of not using `--enable-sp`.

2.8.6.2 --enable-sp or --enable-sp=yes

- WOLFSSL_HAVE_SP_RSA
- WOLFSSL_HAVE_SP_ECC
- WOLFSSL_HAVE_SP_DH
- WOLFSSL_SP_384
- WOLFSSL_SP_521
- WOLFSSL_SP_4096
- WOLFSSL_SP_LARGE_CODE

(WOLFSSL_SP_384, WOLFSSL_SP_521 and WOLFSSL_SP_4096 are added on 64-bit Intel/ARM targets by default; see the note above. For other targets, they should be added manually.)

2.8.6.3 --enable-sp=small

- WOLFSSL_HAVE_SP_RSA
- WOLFSSL_HAVE_SP_DH
- WOLFSSL_HAVE_SP_ECC
- WOLFSSL_SP_4096
- WOLFSSL_SP_384
- WOLFSSL_SP_4096
- WOLFSSL_SP_SMALL
- WOLFSSL_SP_LARGE_CODE
- WOLFSSL_SP_384

2.8.6.4 --enable-sp=smallfast

- WOLFSSL_HAVE_SP_RSA
- WOLFSSL_HAVE_SP_DH
- WOLFSSL_HAVE_SP_ECC
- WOLFSSL_SP_4096
- WOLFSSL_SP_384
- WOLFSSL_SP_SMALL
- WOLFSSL_SP_4096
- WOLFSSL_SP_LARGE_CODE
- WOLFSSL_SP_FAST_MODEXP

2.8.6.5 --enable-sp=ec256 or --enable-sp=p256 or --enable-sp=p256

- WOLFSSL_HAVE_SP_ECC

2.8.6.6 --enable-sp=smallec256 or --enable-sp=smallp256 or --enable-sp=small256

- WOLFSSL_HAVE_SP_ECC
- WOLFSSL_SP_SMALL

2.8.6.7 --enable-sp=ec384 or --enable-sp=p384 or --enable-sp=384

- WOLFSSL_HAVE_SP_ECC
- WOLFSSL_SP_384
- WOLFSSL_SP_NO_256

2.8.6.8 --enable-sp=smallec384 or --enable-sp=smallp384 or --enable-sp=small384

- WOLFSSL_HAVE_SP_ECC
- WOLFSSL_SP_384
- WOLFSSL_SP_NO_256
- WOLFSSL_SP_SMALL

2.8.6.9 --enable-sp=ec1024 or --enable-sp=p1024 or --enable-sp=1024

- WOLFSSL_HAVE_SP_ECC
- WOLFSSL_SP_1024
- WOLFSSL_SP_NO_256

2.8.6.10 `--enable-sp=smallec1024` or `--enable-sp=smallp1024` or `--enable-sp=small1024`

- WOLFSSL_HAVE_SP_ECC
- WOLFSSL_SP_1024
- WOLFSSL_SP_NO_256
- WOLFSSL_SP_SMALL

2.8.6.11 `--enable-sp=2048`

- WOLFSSL_HAVE_SP_DH
- WOLFSSL_HAVE_SP_RSA
- WOLFSSL_SP_LARGE_CODE
- WOLFSSL_SP_NO_3072

2.8.6.12 `--enable-sp=small2048`

- WOLFSSL_HAVE_SP_DH
- WOLFSSL_HAVE_SP_RSA
- WOLFSSL_SP_LARGE_CODE
- WOLFSSL_SP_NO_3072
- WOLFSSL_SP_SMALL

2.8.6.13 `--enable-sp=rsa2048`

- WOLFSSL_HAVE_SP_RSA
- WOLFSSL_SP_LARGE_CODE
- WOLFSSL_SP_NO_3072

2.8.6.14 `--enable-sp=smallrsa2048`

- WOLFSSL_HAVE_SP_RSA
- WOLFSSL_SP_LARGE_CODE
- WOLFSSL_SP_NO_3072
- WOLFSSL_SP_SMALL

2.8.6.15 `--enable-sp=3072`

- WOLFSSL_HAVE_SP_DH
- WOLFSSL_HAVE_SP_RSA
- WOLFSSL_SP_LARGE_CODE
- WOLFSSL_SP_NO_2048

2.8.6.16 `--enable-sp=small3072`

- WOLFSSL_HAVE_SP_DH
- WOLFSSL_HAVE_SP_RSA
- WOLFSSL_SP_LARGE_CODE
- WOLFSSL_SP_NO_2048
- WOLFSSL_SP_SMALL

2.8.6.17 `--enable-sp=rsa3072`

- WOLFSSL_HAVE_SP_RSA
- WOLFSSL_SP_LARGE_CODE
- WOLFSSL_SP_NO_2048

2.8.6.18 --enable-sp=smallrsa3072

- WOLFSSL_SP_LARGE_CODE
- WOLFSSL_SP_NO_2048
- WOLFSSL_SP_SMALL

2.8.6.19 --enable-sp=4096 or --enable-sp=+4096

- WOLFSSL_HAVE_SP_DH
- WOLFSSL_HAVE_SP_RSA
- WOLFSSL_SP_4096
- WOLFSSL_SP_LARGE_CODE
- WOLFSSL_SP_NO_2048
- WOLFSSL_SP_NO_3072

2.8.6.20 --enable-sp=small4096

- WOLFSSL_HAVE_SP_DH
- WOLFSSL_HAVE_SP_RSA
- WOLFSSL_SP_4096
- WOLFSSL_SP_LARGE_CODE
- WOLFSSL_SP_NO_2048
- WOLFSSL_SP_NO_3072
- WOLFSSL_SP_SMALL

2.8.6.21 --enable-sp=rsa4096

- WOLFSSL_HAVE_SP_RSA
- WOLFSSL_SP_4096
- WOLFSSL_SP_LARGE_CODE
- WOLFSSL_SP_NO_2048
- WOLFSSL_SP_NO_3072

2.8.6.22 --enable-sp=smallrsa4096

- WOLFSSL_HAVE_SP_RSA
- WOLFSSL_SP_4096
- WOLFSSL_SP_LARGE_CODE
- WOLFSSL_SP_NO_2048
- WOLFSSL_SP_NO_3072
- WOLFSSL_SP_SMALL

2.8.6.23 --enable-sp=nomalloc

- WOLFSSL_SP_NO_MALLOC

2.8.6.24 --enable-sp=nonblock

- WOLFSSL_SP_NO_MALLOC
- WOLFSSL_SP_NONBLOCK
- WOLFSSL_SP_SMALL

2.8.6.25 asm Combine with other algorithm options to indicate that assembly code is turned on for those options. For example, `--enable-sp=rsa2048,asm`.

2.9 Cross Compiling

Many users on embedded platforms cross compile wolfSSL for their environment. The easiest way to cross compile the library is to use the `./configure` system. It will generate a Makefile which can then be used to build wolfSSL.

When cross compiling, you'll need to specify the host to `./configure`, such as:

```
./configure --host=arm-linux
```

You may also need to specify the compiler, linker, etc. that you want to use:

```
./configure --host=arm-linux CC=arm-linux-gcc AR=arm-linux-ar RANLIB=arm-linux
```

There is a bug in the configure system which you might see when cross compiling and detecting user overriding malloc. If you get an undefined reference to `rpl_malloc` and/or `rpl_realloc`, please add the following to your `./configure` line:

```
ac_cv_func_malloc_0_nonnull=yes ac_cv_func_realloc_0_nonnull=yes
```

After correctly configuring wolfSSL for cross-compilation, you should be able to follow standard autoconf practices for building and installing the library:

```
make
sudo make install
```

If you have any additional tips or feedback about cross compiling wolfSSL, please let us know at facts@wolfssl.com.

2.9.1 Example cross compile configure options for toolchain builds

2.9.1.1 armebv7-eabihf-glibc

```
./configure --host=armeb-linux \
        CC=armeb-linux-gcc LD=armeb-linux-ld \
        AR=armeb-linux-ar \
        RANLIB=armeb-linux-ranlib \
        CFLAGS="-DWOLFSSL_USER_IO -Os" \
        CPPFLAGS="-I./"
```

2.9.1.2 armv5-eabi-glibc

```
./configure --host=arm-linux \
        CC=arm-linux-gcc LD=arm-linux-ld \
        AR=arm-linux-ar \
        RANLIB=arm-linux-ranlib \
        CFLAGS="-DWOLFSSL_USER_IO -Os" \
        CPPFLAGS="-I./"
```

2.9.1.3 armv6-eabihf-glibc

```
./configure --host=arm-linux \
        CC=arm-linux-gcc LD=arm-linux-ld \
        AR=arm-linux-ar \
        RANLIB=arm-linux-ranlib \
        CFLAGS="-DWOLFSSL_USER_IO -Os" \
        CPPFLAGS="-I./"
```

2.9.1.4 armv7-eabihf-glibc

```
./configure --host=arm-linux \
        CC=arm-linux-gcc LD=arm-linux-ld \
        AR=arm-linux-ar \
        RANLIB=arm-linux-ranlib \
        CFLAGS="-DWOLFSSL_USER_IO -Os" \
        CPPFLAGS="-I./"
```

2.9.1.5 armv7m-uclibc

```
./configure --enable-static --disable-shared \--host=arm-linux
 ↪  CC=arm-linux-gcc \
        LD=arm-linux-ld AR=arm-linux-ar \
        RANLIB=arm-linux-ranlib \
        CFLAGS="-DWOLFSSL_USER_IO -Os" \
        CPPFLAGS="-I./"
```

2.9.1.6 arm-none-eabi-gcc

```
./configure --host=arm-none-eabi \
        CC=arm-none-eabi-gcc LD=arm-none-eabi-ld \
        AR=arm-none-eabi-ar RANLIB=arm-none-eabi-ranlib \
        CFLAGS="-DNO_WOLFSSL_DIR \
        -DWOLFSSL_USER_IO -DNO_WRITEV \
        -mcpu=cortex-m4 -mthumb -Os \
        -specs=rdimon.specs" CPPFLAGS="-I./"
```

2.9.1.7 mips32–glibc

```
./configure --host=mips-linux \
        CC=mips-linux-gcc LD=mips-linux-ld \
        AR=mips-linux-ar \
        RANLIB=mips-linux-ranlib \
        CFLAGS="-DWOLFSSL_USER_IO -Os" \
        CPPFLAGS="-I./"
```

2.9.1.8 PowerPc64le-Power8-Glibc

```
./configure --host=powerpc64le-buildroot-linux-gnu \
        CC=powerpc64le-buildroot-linux-gnu-gcc \
        LD=powerpc64le-buildroot-linux-gnu-ld \
        AR=powerpc64le-buildroot-linux-gnu-ar \
        RANLIB=powerpc64le-buildroot-linux-gnu-ranlib \
        CFLAGS="-DWOLFSSL_USER_IO -Os" \
        CPPFLAGS="-I./"
```

2.9.1.9 x86-64-core-i7-glibc

```
./configure --host=x86_64-linux \
        CC=x86_64-linux-gcc LD=x86_64-linux-ld \
        AR=x86_64-linux-ar \
        RANLIB=x86_64-linux-ranlib \
        CFLAGS="-DWOLFSSL_USER_IO -Os" \
        CPPFLAGS="-I./"
```

2.9.1.10 x86-64-core-i7-musl

```
./configure --host=x86_64-linux \
        CC=x86_64-linux-gcc LD=x86_64-linux-ld \
        AR=x86_64-linux-ar \
        RANLIB=x86_64-linux-ranlib \
        CFLAGS="-DWOLFSSL_USER_IO -Os" \CPPFLAGS="-I./"
```

2.9.1.11 x86-64-core-i7-uclibc

```
./configure --host=x86_64-linux \
        CC=x86_64-linux-gcc LD=x86_64-linux-ld \
        AR=x86_64-linux-ar \
        RANLIB=x86_64-linux-ranlib \
        CFLAGS="-DWOLFSSL_USER_IO -Os" \
        CPPFLAGS="-I./"
```

2.10 Building Ports

wolfSSL has been ported to many environments and devices. A portion of these ports and accompanying documentation for them is located in the directory `wolfssl-X.X.X/IDE`, where X.X.X is the current wolfSSL version number. This directory also contains helpful information and code for IDE's used to build wolfSSL for the environments.

PORT Lists:

- Arduino
- LPCXPRESSO
- Wiced Studio
- CSBench
- SGX Windows and Linux
  - These directories (`wolfssl/IDE/WIN-SGX` and `wolfssl/IDE/LINUX-SGX`) contain Makefiles for and Visual Studio solutions for building wolfSSL as a library to be used in an Intel SGX project.
- Hexagon
  - This directory (`wolfssl/IDE/HEXAGON`) contains a Makefile for building with the Hexagon tool chain. It can be used to build wolfSSL for offloading ECC verify operations to a DSP processor. The directory contains a README file to help with the steps required to build.
- Hexiwear
- NetBurner M68K
  - In the directory (`wolfssl/IDE/M68K`) there is a Makefile for building wolfSSL for a MCF5441X device using the Netburner RTOS.
- Renesas
  - This directory (`wolfssl/IDE/Renesas`) contains multiple builds for different Renesas devices. It also has example builds that demonstrate using hardware acceleration.
- XCode
- Eclipse
- Espressif
- IAR-EWARM
- Kinetis Design Studio (KDS)
- Rowley Crossworks ARM
- OpenSTM32
- RISCV
- Zephyr
- Mynewt

- INTIME-RTOS

2.11 Building For NXP CAAM

2.11.1 i.MX8 (Linux)

2.11.1.1 Known Issues

- If exiting an open HSM key store session before closing up the HSM session (wc_SECO_CloseHSM and wolfSSL_CLeanup or wolfCrypt_Cleanup) the next time the NVM is started up it segfaults. A power cycle is needed to work around it.

2.11.1.2 Limitations Found

- AES operations with large inputs (i.e. 1 megabyte) fails with the SECO getting “Not enough space in shared memory”.
- After creating 2 keystores the attempt to create a 3rd will fail. To reset the keystores do rm -rf /etc/seco_hsm and power cycle the device.

2.11.1.3 Intro On i.MX8 devices there is a SECO hardware module available for heightened security. This module handles AES operations and key storage, limited ECC operations and key storage, and provides a RNG. wolfSSL has been expanded to make use of the SECO where possible. For some algorithms the CAAM on i.MX8 supports them but the SECO module does not yet. In these cases wolfSSL will make calls through /dev/crypto down to a Linux CAAM driver that creates jobs for the CAAM directly. There are some algorithms supported by default with the NXP Linux CAAM driver but not all algorithms that the CAAM supports. wolfSSL has expanded the Linux CAAM driver to add support for additional algorithms. To use both avenues of accessing the CAAM from the same application the “devId” associated with the different code paths can be set, either WOLFSSL_CAAM_DEVID or WOLFSSL_SECO_DEVID. These IDs are used when first initializing a structure to set which code path will be used throughout the lifetime of the structure. If using software only then the default INVALID_DEVID should be set. The exception to this is SECO items that do not use the key store; TRNG, and Hashing.

Versions of software used:

- imx-seco-libs branch imx_5.4.24_2.1.0
- NXP “repo” tool and Yocto build. Documentation on Yocto setup is here
- wolfSSL 5.2.0 + (was developed after 5.2.0 release)

2.11.1.4 Supported Algorithms Supported algorithms, modes, and operations include:

- AES (ECB, CBC, CCM, GCM)
- AES CMAC
- SHA256, SHA384
- ECC 256/384 (keygen, sign, verify, ecdh)
- RSA 3072 (keygen, sign, verify)
- HMAC
- Curve25519
- TRNG

2.11.1.5 Building Image

2.11.1.5.1 “repo” Setup Setting up the NXP “repo” command tool was done on Ubuntu 18.04 LTS

```
sudo apt-get install gawk wget git-core diffstat unzip texinfo gcc-multilib
sudo apt-get install build-essential chrpath socat cpio python python3 python3
    -pip
sudo apt-get install python3-pexpect xz-utils debianutils iputils-ping python3
    -git
sudo apt-get install python3-jinja2 libegl1-mesa libsdl1.2-dev pylint3 xterm
    curl
sudo apt-get install ca-certificates

mkdir ~/bin
curl https://storage.googleapis.com/git-repo-downloads/repo > ~/bin/repo
chmod a+x ~/bin/repo
export PATH=~/bin:$PATH

git config --global user.name "Your Git Name"
git config --global user.email "Your Email"
```

As of Jan 11, 2022 GitHub is no longer allowing unauthenticated git connections. NXP's repo tool has not yet taken this into account at the time that this document was created (March, 2022). To work around this redirect git://github.com/ to https://github.com/ with the following command:

```
git config --global url."https://github.com/".insteadOf git://github.com/
```

Make the desired directory to build in, for example:

```
mkdir imx-yocto-bsp
cd imx-yocto-bsp/
```

After setting up NXP's "repo" command tool, initialize and sync a directory with the desired version of Linux. In this case 5.4.24_2.1.0.

```
repo init -u https://source.codeaurora.org/external/imx/imx-manifest -b imx-
    linux-zeus -m imx-5.4.24-2.1.0.xml
repo sync

DISTRO=fsl-imx-wayland MACHINE=imx8qxpc0mek source imx-setup-release.sh -b
    build-xwayland
```

2.11.1.5.2 Additional Yocto CAAM Layers Next download the CAAM driver expansion layers that will apply patches to cryptodev-module, cryptodev-linux, and to files in `linux-imx/drivers/crypto/caam/*`. The base blob and ECDSA (sign/verify/keygen) is located here (https://source.codeaurora.org/external/imxsupport/imx The layer that expands that farther to have RSA black key, ECDH, and Curve25519 support is meta-imx-expand-caam. Place both of these directories into the sources directory next to the other existing meta-* directories.

```
#<assuming in the build-xwayland directory from previous command>

#<either clone the current work or open from a delivered zip>
git clone -b caam_expansion https://github.com/JacobBarthelmeh/imx_sec_apps
cp -r imx_sec_apps/meta-imx-ecdsa-sec ../sources/
cp -r imx_sec_apps/meta-imx-expand-caam ../sources/


# or
```

```
git clone https://source.codeaurora.org/external/imxsupport/imx_sec_apps.git
#(meta-imx-expand-caam comes from wolfSSL)
unzip meta-imx-expand-caam.zip

cp -r imx_sec_apps/meta-imx-ecdsa-sec ../sources/
mv meta-imx-expand-caam ../sources/
```

Add those layers to the build, ecdsa first, then the CAAM expansion.

```
vim conf/bblayers.conf
BBLAYERS += "${BSPDIR}/sources/meta-imx-ecdsa-sec"
BBLAYERS += "${BSPDIR}/sources/meta-imx-expand-caam"
```

Add the desired modules and libraries to the local conf file.

```
vim conf/local.conf
EXTRA_IMAGE_FEATURES_append = " dev-pkgs tools-sdk tools-debug
 ↪  ssh-server-openssh "
IMAGE_INSTALL_append = " cryptodev-module cryptodev-linux eckey "
```

In this build, we had added debugging tools and an SSH server, which are not necessary if looking to trim down on size. The important items to add are “cryptodev-module and cryptodev-linux”. “eckey” is a demo tool from NXP for encapsulating and decapsulating blobs.

[Optional] To add the auto loading of cryptodev module add the following line to conf/local.conf.

```
KERNEL_MODULE_AUTOLOAD += "cryptodev"
```

Otherwise the module will need to be loaded after each power cycle using “modprobe cryptodev”.

2.11.1.5.3 Build and Deploy To kick off the build of the image use the following command. Then if using an sdcard, flash it to the card.

```
bitbake core-image-base

cd tmp/deploy/images/imx8qxpc0mek/
bzcat core-image-base-imx8qxpc0mek.sdcard.bz2 | sudo dd of=/dev/diskX bs=5m
```

Note: 5m is for Mac OS, use 5M for Linux. diskX should be replaced with the location of sdcard, i.e. disk2 on Mac or sdbX on Linux. Check to make sure of the sdcard disk number before executing. For use with building wolfssl/examples later export the install directory using the following:

```
exportCRYPTODEV_DIR=`pwd`/tmp/sysroots-components/aarch64/cryptodev-linux/usr/
   include/
```

To install the toolchain for cross compiling use the following Yocto command

```
bitbake meta-toolchain
sudo ./tmp/deploy/sdk/<version>.sh
```

2.11.1.6 Building NXP HSM

2.11.1.6.1 Build zlib Multiple ways to do this, one is to add it to the Yocto build one way is to build it using bitbake as follows.

```
cd build-xwayland
bitbake zlib
```

This places the results into the directory tmp/sysroots-components/aarch64/zlib/usr/

For use with building wolfssl/examples later export the install directory using the following:

```
export ZLIB_DIR=`pwd`/tmp/sysroots-components/aarch64/zlib/usr/
```

2.11.1.6.2 Build NXP HSM lib Download the NXP HSM library, and adjust the Makefile (or environment variables) in order to find the necessary zlib.

```
git clone https://github.com/NXP/imx-seco-libs.git
cd imx-seco-libs
git checkout imx_5.4.24_2.1.0
```

```
vim Makefile
```

```
CFLAGS = -O1 -Werror -fPIC -I$(ZLIB_DIR)/include -L$(ZLIB_DIR)/lib
```

then

```
make
make install
```

For use with building wolfssl/examples later export the install directory using the following:

```
export HSM_DIR=`pwd`/export/usr/
```

Make install places the results in the subdirectory “export” by default.

2.11.1.7 Building wolfSSL

2.11.1.7.1 Building Using Autoconf If setting up the Yocto image with development tools then wolfSSL can be built directly on the system. For a more minimalistic approach cross compiling can be used. Debug messages can be enabled with –enable-debug. Extra debug messages specific to the SECO work can be enabled by defining the macro DEBUG_SECO and for the /dev/crypto calls DEBUG_DEVCRYPTO. The extra debug messages for both make use of printf, outputting on the stdout pipe. There are a couple key enable options for use with SECO. –enable-caam=seco, –enable-devcrypto=seco, –with-seco=/hsm-lib/export.

Example build with HSM SECO only (no devcrypto support for additional algorithms)

```
source /opt/fsl-imx-wayland/5.4-zeus/environment-setup-aarch64-poky-linux

# Install dependencies for building wolfSSL
sudo apt-get install autoconf automake libtool

./autogen.sh
./configure --host=aarch64-poky-linux --with-libz=$ZLIB_DIR
 ↪  --with-seco=$HSM_DIR \ --enable-caam=seco --enable-cmac --enable-aesgcm
 ↪  --enable-aesccm --enable-keygen \
CPPFLAGS="-DHAVE_AES_ECB"
make
```

Example building with HSM SECO and additional devcrypto support. The include path to crypto/cryptodev.h needs to be set.

```
./configure --host=aarch64-poky-linux --with-libz=$ZLIB_DIR
 ↪  --with-seco=$HSM_DIR \
--enable-caam=seco --enable-cmac --enable-aesgcm --enable-aesccm
 ↪  --enable-keygen \
CPPFLAGS="-DHAVE_AES_ECB -I$CRYPTODEV_DIR" --enable-devcrypto=seco \
--enable-curve25519
make
```

There are fail safes to error out early on wolfCrypt_Init / wolfSSL_Init function calls. One case is if the cryptodev module has not been loaded or does not have the support available for the desired operations. The other case where the init operation would fail is if the NXP HSM was not able to be set up. If the application is failing on initialization, adding –enable-debug to the wolfSSL build and the function call wolfSSL_Debugging_ON() before the initialization of wolfSSL will print out useful debug messages about why it is failing.

Example building with debug options turned on

```
./configure --host=aarch64-poky-linux --with-libz=$ZLIB_DIR
 ↪  --with-seco=$HSM_DIR \
--enable-caam=seco --enable-cmac --enable-aesgcm --enable-aesccm
 ↪  --enable-keygen \
CPPFLAGS="-DHAVE_AES_ECB -I$CRYPTODEV_DIR -DDEBUG_SECO -DDEBUG_DEVCRYPTO" \
--enable-devcrypto=seco --enable-curve25519
```

2.11.1.7.2 Building Using user_settings.h These are the macros that can be enabled for building without autotools:

CAAM

- WOLFSSL_CAAM - Main macro switch to enable CAAM support.
- WOLF_CRYPTO_CB - CAAM support makes use of crypto callbacks
- WOLFSSL_SECO_CAAM - Enable SECO HSM use with CAAM (AES-GCM is required and used as the algo for importing plain keys into the HSM).
- WOLFSSL_HASH_KEEP - When hashing with algos like SHA256 build up the message and send it to be hashed only on a call to Final.
- WOLFSSL_CAAM_ECC - Enable CAAM ECC support.
- WOLFSSL_CAAM_CMAC - Enable CAAM CMAC support.
- WOLFSSL_CAAM_CIPHER - Enable CAAM AES support.
- WOLFSSL_CAAM_HMAC - Enable CAAM HMAC support.
- WOLFSSL_CAAM_HASH - Enable CAAM hashing support such as SHA256.
- WOLFSSL_CAAM_CURVE25519 - Enable CAAM Curve25519 support.

cryptodev-linux

- WOLFSSL_DEVCRYPTO - Main macro switch to enable cryptodev-linux use.
- WOLFSSL_DEVCRYPTO_HMAC - Enable support of HMAC with cryptodev-linux.
- WOLFSSL_DEVCRYPTO_RSA - Enable support of RSA with cryptodev-linux.
- WOLFSSL_DEVCRYPTO_CURVE25519 - Enable support of Curve25519 with cryptodev-linux.
- WOLFSSL_DEVCRYPTO_ECDSA - Enable support of ECDSA with cryptodev-linux.
- WOLFSSL_DEVCRYPTO_HASH_KEEP - Enable support of storing up hashes with cryptodev-linux.

Additional files that need compiled for CAAM support are:

- wolfssl/wolfcrypt/src/port/caam/wolfcaam_aes.c
- wolfssl/wolfcrypt/src/port/caam/wolfcaam_cmac.c
- wolfssl/wolfcrypt/src/port/caam/wolfcaam_rsa.c
- wolfssl/wolfcrypt/src/port/caam/wolfcaam_ecdsa.c
- wolfssl/wolfcrypt/src/port/caam/wolfcaam_x25519.c

- wolfssl/wolfcrypt/src/port/caam/wolfcaam_hash.c
- wolfssl/wolfcrypt/src/port/caam/wolfcaam_hmac.c
- wolfssl/wolfcrypt/src/port/caam/wolfcaam_init.c
- wolfssl/wolfcrypt/src/port/caam/wolfcaam_seco.c
- wolfssl/wolfcrypt/src/port/devcrypto/devcrypto_ecdsa.c
- wolfssl/wolfcrypt/src/port/devcrypto/devcrypto_x25519.c
- wolfssl/wolfcrypt/src/port/devcrypto/devcrypto_rsa.c
- wolfssl/wolfcrypt/src/port/devcrypto/devcrypto_hmac.c
- wolfssl/wolfcrypt/src/port/devcrypto/devcrypto_hash.c
- wolfssl/wolfcrypt/src/port/devcrypto/devcrypto_aes.c
- wolfssl/wolfcrypt/src/port/devcrypto/wc_devcrypto.c
- wolfssl/wolfcrypt/src/cryptocb.c

2.11.1.8 Examples

2.11.1.8.1 Running Testwolfcrypt The unit tests that are bundled with wolfSSL are located in wolfcrypt/test/test.c. An example of building and running the tests on the device would be the following. Note that this uses WOLFSSL_CAAM_DEVID so it is making use of the cryptodev module and not the NXP HSM library.

```
./configure --host=aarch64-poky-linux --with-libz=$ZLIB_DIR \
--with-seco=$HSM_DIR --enable-caam=seco --enable-cmac --enable-aesgcm \
--enable-aesccm --enable-keygen CPPFLAGS="-DHAVE_AES_ECB -I$CRYPTODEV_DIR" \
--enable-devcrypto=seco --enable-curve25519 --enable-sha224 --enable-static \
--disable-shared --disable-filesystem
make

scp wolfcrypt/test/testwolfcrypt root@192.168.0.14:/tmp
ssh root@192.168.0.14
root@imx8qxpc0mek:~# /tmp/testwolfcrypt
------------------------------------------------------------------------------
 wolfSSL version 5.2.0
------------------------------------------------------------------------------
error   test passed!
MEMORY   test passed!
base64   test passed!
asn     test passed!
RANDOM   test passed!
MD5     test passed!
SHA     test passed!
SHA-224  test passed!
SHA-256  test passed!
SHA-384  test passed!
SHA-512  test passed!
SHA-3   test passed!
Hash    test passed!
HMAC-MD5 test passed!
HMAC-SHA test passed!
HMAC-SHA224 test passed!
HMAC-SHA256 test passed!
HMAC-SHA384 test passed!
HMAC-SHA512 test passed!
HMAC-SHA3   test passed!
HMAC-KDF    test passed!
```

```
GMAC    test passed!
Chacha   test passed!
POLY1305 test passed!
ChaCha20-Poly1305 AEAD test passed!
AES     test passed!
AES192   test passed!
AES256   test passed!
AES-GCM  test passed!
AES-CCM  test passed!
RSA     test passed!
DH      test passed!
PWDBASED test passed!
ECC     test passed!
ECC buffer test passed!
CURVE25519 test passed!
CMAC    test passed!
COMPRESS test passed!
logging  test passed!
time test passed!
mutex   test passed!
memcb   test passed!
crypto callback test passed!
Test complete
Exiting main with return code: 0
root@imx8qxpc0mek:~#
```

Additional examples are located in the wolfssl-examples repository under the caam/seco directory.

```
git clone https://github.com/wolfssl/wolfssl-examples
cd wolfssl-examples/caam/seco
make
```

2.11.1.8.2 Compiling Source Code [using user_settings.h] To build a single source file linking to wolfSSL the following commands could be used. Assuming that the env. variables from the previous steps are still set.

```
source /opt/fsl-imx-xwayland/5.4-zeus/environment-setup-aarch64-poky-linux

$CC -DWOLFSSL_USER_SETTINGS -I /path/to/user_settings.h \
-I $CRYPTODEV_DIR -I $HSM_DIR/include -I ./wolfssl server-dtls.c \
libwolfssl.a $HSM_DIR/lib/hsm_lib.a $HSM_DIR/lib/seco_nvm_manager.a \
$ZLIB_DIR/lib/libz.a -lpthread -lm
```

2.11.1.9 API

2.11.1.9.1 API Added *List of additional API added*

- void wc_SECO_AesSetKeyID(Aes* aes, int keyId); This function is used to set a SECO key ID into an Aes structure. It should be called after the Aes structure has been initialized and before the structure gets used for encrypt/decrypt operations.
- int wc_SECO_AesGetKeyID(Aes* aes); A getter function for the SECO key ID set in the Aes structure.
- void wc_SECO_CMACSetKeyID(Cmac* cmac, int keyId); Similar to wc_SECO_AesSetKeyID but for Cmac structures.

- int wc_SECO_CMACGetKeyID(Cmac* cmac); Getter function for the SECO key ID set in the Cmac structure.
- int wc_SECO_OpenHSM(word32 keyStoreId, word32 nonce, word16 maxUpdates, byte flag); This function should be called before doing any operations that require the keystore. Such as ECC or AES operations. The first argument is the keystore ID, "nonce" is the specific 32bit sequence used when creating and unlocking an already existing keystore, "maxUpdates" sets the max number of times the key store can be updated, "flag" is used for optional flags and takes in whether or not the key store is being created. To create the HSM keystore the flag should be CAAM_KEYSTORE_CREATE otherwise to open an existing keystore or update it the flag should be CAAM_KEYSTORE_UPDATE.
- int wc_SECO_CloseHSM(void); This function must be called when done with the keystore and before calling wolfCrypt_Cleanup/wolfSSL_Cleanup. It closes down the currently open keystore.
- int wc_SECO_GenerateKey(int flags, int group, byte* out, int outSz, int keyType, int keyInfo, unsigned int* keyIdOut);
  This function can be used for generating new keys in the SECO. For key generation the flag should be CAAM_GENERATE_KEY. For updating a key the flag should be CAAM_UPDATE_KEY. The keyIdOut argument is an in/out argument that gets populated on key creation and should be set as an input on key update. In the case of updating a key it should be a transient type and the group should be set to 0 for updating. Transient type is set as the keyInfo arg, possible options for keyInfo and KeyTypes are;
  - CAAM_KEY_TRANSIENT (keyInfo)
  - CAAM_KEY_PERSISTENT (keyInfo)
  - CAAM_KEYTYPE_ECDSA_P256 (keyType)
  - CAAM_KEYTYPE_ECDSA_P384 (keyType)
  - CAAM_KEYTYPE_AES128 (keyType)
- int wc_SECO_DeleteKey(unsigned int keyId, int group, int keyTypeIn); Used to delete a key from the keystore.

2.11.1.9.2 Native wolfSSL API With CAAM Support This is a list of native wolfSSL API that now have CAAM support with the SECO build outlined in this documentation.

For generation of any AES encrypt and decrypt operations the key can be generated using the following process. Using wc_SECO_GenerateKey(CAAM_GENERATE_KEY, groupID, pubOut, 0, CAAM_KEYTPE_AES128, CAAM_KEY_PERSISTENT, &keyIdOut); where groupID is a specified group number and pubOut is a 32 byte buffer, and the variable keyIdOut gets set to a the new key ID generated. This new key ID generated can then be set in an Aes structure using wc_SECO_AesSetKeyID(Aes, keyIdOut); . Once the key ID has been set in the structure and the Aes structure has been initialized as a WOLFSSL_SECO_DEVID type it will use that key ID for all encrypt and decrypt operations.

AES (ECB/CBC)

Alternatively to generate AES ECB/CBC keys, if the Aes structure has been initialized with WOLFSSL_SECO_DEVID the function wc_AesSetKey can be called with a plain text key passed to it. The API wc_AesSetKey will then try to encrypt the key using the unique KEK and import it into the SECO HSM. If imported successfully, the value of 0 will be returned and the key ID will be set in the Aes structure.

- CBC encrypt would be done with wc_AesCbcEncrypt, decrypt with wc_AesCbcDecrypt.
- ECB encrypt would be done with wc_AesEcbEncrypt, decrypt with wc_AesEcbDecrypt.

Once finished with the Aes structure it must be free'd using wc_AesFree(Aes); .

AES-GCM

- GCM encrypt would be done with wc_AesGcmEncrypt, decrypt with wc_AesGcmDecrypt.

The AES-GCM encrypt function takes in the Aes structure, output buffer, input buffer, input buffer size, nonce, nonce size (required to be 12 bytes), MAC or known as tag, tag size (required to be 16 bytes),

additional data, additional data size (4 bytes). On encryption the input buffer is encrypted and the tag buffer is filled with a created MAC.

For AES-GCM decrypt the function takes in the Aes structure, plain text output buffer, cipher text input buffer, input buffer size, nonce, nonce size (12 bytes), previously created tag from encryption call, tag buffer size, additional data, additional data size. On decryption the tag buffer is checked to verify the message's integrity.

Once finished with the Aes structure it must be free'd using wc_AesFree(Aes); .

AES-CCM

- CCM encrypt would be done with wc_AesCcmEncrypt, decrypt with wc_AesCcmDecrypt.

The AES-CCM encrypt function takes in the Aes structure, output buffer, input buffer, input buffer size, nonce, nonce size (required to be 12 bytes), MAC or known as tag, tag size (required to be 16 bytes), additional data, additional data size (0 bytes). The additional data buffer should be NULL and a size of 0 is required with the NXP HSM library. On encryption the input buffer is encrypted and the tag buffer is filled with a created MAC.
For AES-CCM decrypt the function takes in the Aes structure, plain text output buffer, cipher text input buffer, input buffer size, nonce, nonce size (12 bytes), previously created tag from encryption call, tag buffer size, additional data, additional data size. Similar to the encrypt function the additional data buffer should be NULL. On decryption the tag buffer is checked to verify the message's integrity.
Once finished with the Aes structure it must be free'd using wc_AesFree(Aes); .

AES CMAC

For AES CMAC operations the AES key can be generated using wc_SECO_GenerateKey(CAAM_GENERATE_KEY, groupID, pubOut, 0, CAAM_KEYTPE_AES128, CAAM_KEY_PERSISTENT, &keyIdOut); where groupID is a specified group number and pubOut is a 32 byte buffer, and the variable keyIdOut gets set to a the new key ID generated. This new key ID generated can then be set in an Aes structure using wc_SECO_CMACSetKeyID(Cmac, keyIdOut); . Once the key ID has been set in the structure and the Aes structure has been initialized as a WOLFSSL_SECO_DEVID type it will use that key ID for all encrypt and decrypt operations.

Since the HSM library is a single shot type, each call to wc_CmacUpdate stores up the input into an internal buffer. Then once wc_CmacFinal is called the whole buffer is passed on to the hardware for creating the MAC.

RSA

RSA operations make use of the cryptodev-linux module. This includes support for AES-ECB encrypted black private keys, which is the default when initialized with WOLFSSL_CAAM_DEVID.

Example of native wolfSSL API that would be used with the cryptodev-linux module is as follows:

```
wc_InitRsaKey_ex(key, heap-hint (can be NULL), WOLFSSL_CAAM_DEVID);
wc_MakeRsaKey(key, 3072, WC_RSA_EXPONENT, &rng);
wc_RsaSSL_Sign or wc_RsaPublicEncrypt
wc_RsaSSL_Verify or wc_RsaPrivateDecrypt
wc_FreeRsaKey(key)
```

ECC

ECC sign and verify operations can use either the cryptodev-linux module or the NXP HSM library. ECDH operations for creating a shared secret can only be done with the cryptodev-linux module.

For use with SECO (using the NXP HSM library) the dev ID flag of WOLFSSL_SECO_DEVID should be used when initializing the ecc_key structure. For use with the cryptodev-linux module the dev ID flag

WOLFSSL_CAAM_DEVID should be used. After initialization with the function wc_ecc_init_ec(key, heap-hint (can be NULL), dev ID); then both use cases follow the same function native wolfSSL function calls for sign and verify.

Example function calls after initialization of the ecc_key structure would be:

```
wc_ecc_make_key(&rng, ECC_P256_KEYSIZE, key);
wc_ecc_sign_hash(hash, hashSz, sigOut, sigOutSz, &rng, key);
wc_ecc_verify_hash(sig, sigSz, hash, hashSz, &result, key);
```

And with the cryptodev-linux module (WOLFSSL_CAAM_DEVID) the ECDH function can be used:

```
wc_ecc_shared_secret(keyA, keyB, sharedSecret, sharedSecretSz);
```

Hash (Sha256, Sha384, HMAC)

SHA256 and SHA384 operations use the NXP HSM library. HMAC operations make use of the cryptodev-linux module.

By default SHA operations try to make use of the NXP HSM library, but explicitly set them to the dev ID WOLFSSL_SECO_DEVID can be used.

```
wc_InitSha256_ex(sha256, heap-hint, WOLFSSL_SECO_DEVID);
wc_InitSha384_ex(sha384, heap-hint, WOLFSSL_SECO_DEVID);
```

Because the NXP HSM library supports a single shot operation for hashing, each call to “update” will store the buffer until a “final” function is called and then pass the whole buffer on to the hardware for creating the hash digest.

Where HMAC makes use of the cryptodev-linux the Hmac structure should be initialized using the dev ID WOLFSSL_CAAM_DEVID.

```
wc_HmacInit(hmac, heap-hint, WOLFSSL_CAAM_DEVID);
```

It then can be used like it typically would be with native wolfSSL API:

```
wc_HmacSetKey(hmac, hash-type, key, keySz);
wc_HmacUpdate(hmac, input, inputSz);
wc_HmacFinal(hmac, digestOut);
```

Curve25519

Curve25519 point multiplication is done using the cryptodev-linux module and should be initialized with the dev ID WOLFSSL_CAAM_DEVID for use with the hardware.

Example API calls would be as follows:

```
wc_curve25519_init_ex(key, heap-hint, WOLFSSL_CAAM_DEVID);
wc_curve25519_make_key(&rng, CURVE25519_KEYSIZE,  key);
wc_curve25519_shared_secret(key, keyB, sharedSecretOut, sharedSecretOutSz);
```

RNG

TRNG for seeding the wolfSSL HASH-DRBG makes use of the NXP HSM library. This is compiled in to the wolfcrypt/src/random.c file when wolfSSL is built with –enable-caam=seco. All RNG initializations in wolfSSL will make use of the TRNG for seeding. Standard RNG API calls would be as follows:

```
wc_InitRng(rng);
wc_RNG_GenerateBlock(rng, output, outputSz);
wc_FreeRng(rng);
```

2.11.2 i.MX8 (QNX)

<Documentation available, not inserted here yet @TODO>

2.11.3 i.MX6 (QNX)

<Documentation available, not inserted here yet @TODO>

2.11.4 IMXRT1170 (FreeRTOS)

Example IDE Setup for use with IMXRT1170 can be found in the directory IDE/MCUEXPRESSO/RT1170

2.11.4.1 Build Steps

- Open MCUEXPRESSO and set the workspace to wolfssl/IDE/MCUEXPRESSO/RT1170
- File -> Open Projects From File System... -> Directory : and set the browse to wolfssl/IDE/MCUEXPRESSO/RT1170 directory then click “select directory”
- Select wolfssl_cm7, wolfcrypt_test_cm7, CSR_example, PKCS7_example
- Right click the projects -> SDK Management -> Refresh SDK Components and click “yes”
- increase the size of configTOTAL_HEAP_SIZE in FreeRTOSConfig.h to be 60240 for CSR and PKCS7 example and around 100000 for wolfcrypt_test_cm7
- (note board files need to be recreated .... this can be done by creating a new project that has the same settings and copying over the generated board/* files)
- Build the projects

2.11.4.2 Expanding RT1170 CAAM Driver The files RT1170/fsl_caam_h.patch and RT1170/fsl_caam_c.patch include changes to the existing NXP CAAM driver for use with creating/opening Blobs and generating and using ECC black keys.

To apply the patches first create a project that has the caam driver. This will generate the base fsl_caam.c and fsl_caam.h in the drivers directory. (i.e PKCS7_example_cm7/drivers/fls_caam.{c,h}) . Once the base files are generated then ‘cd’ to the drivers directory and apply the patch (cd PKCS7_example_cm7/drivers/ && patch -p1 < ../../fsl_caam_c.patch && patch -p1 < ../../fsl_caam_h.patch)

In the patch for fsl_caam.h there are macros defined for both the ECC and Blob expansion (CAAM_ECC_EXPANSION and CAAM_BLOB_EXPANSION). When wolfSSL code finds that these macros are defined (the patch has been applied) then it tries to compile in use of the expanded driver.

2.11.4.3 WOLFSSL_HAVE_ERROR_QUEUE Enables an OpenSSL-compatible error queue for storing and retrieving error information via ERR_`get_error()` and related functions.

2.11.4.4 WOLFSSL_HAVE_CERT_SERVICE Enables certificate service callbacks for custom certificate handling during the TLS handshake.

2.11.4.5 WOLFSSL_HEAP_TEST Enables heap-related testing utilities for verifying memory allocation behavior in wolfSSL.

2.11.4.6 WOLFSSL_NO_OPENSSL_RAND_CB Disables OpenSSL RAND callback compatibility. Prevents the RNG from being overridden via OpenSSL-style RAND_`set_rand_method()` callbacks.

2.11.4.7 WOLFSSL_NO_REALLOC Disables use of `realloc()`. All buffer resizing will be done via `malloc()` + `memcpy()` + `free()` instead.

2.11.4.8 WOLFSSL_DEBUG_DTLS Enables debug logging for DTLS-specific operations including retransmission, epoch management, and record processing.

3 Getting Started

3.1 General Description

wolfSSL, formerly CyaSSL, is about 10 times smaller than yaSSL, and up to 20 times smaller than OpenSSL when using the compile options described in Chapter 2. User benchmarking and feedback also reports dramatically better performance from wolfSSL vs. OpenSSL in the vast majority of standard SSL operations.

For instructions on the build process please see Chapter 2.

3.2 Testsuite

The testsuite program is designed to test the ability of wolfSSL and its cryptography library, wolfCrypt, to run on the system.

wolfSSL needs all examples and tests to be run from the wolfSSL home directory. This is because it finds certs and keys from ./certs. To run testsuite, execute:

```
./testsuite/testsuite.test
```

Or when using autoconf:

```
make test
```

On *nix or Windows the examples and testsuite will check to see if the current directory is the source directory and if so, attempt to change to the wolfSSL home directory. This should work in most setup cases, if not, just use the first method above and specify the full path.

On a successful run you should see output like this, with additional output for unit tests and cipher suite tests:

```
------------------------------------------------------------------------------
 wolfSSL version 4.8.1
------------------------------------------------------------------------------
error    test passed!
MEMORY   test passed!
base64   test passed!
base16   test passed!
asn      test passed!
RANDOM   test passed!
MD5      test passed!
SHA      test passed!
SHA-224  test passed!
SHA-256  test passed!
SHA-384  test passed!
SHA-512  test passed!
SHA-3    test passed!
Hash     test passed!
HMAC-MD5 test passed!
HMAC-SHA test passed!
HMAC-SHA224 test passed!
HMAC-SHA256 test passed!
HMAC-SHA384 test passed!
HMAC-SHA512 test passed!
HMAC-SHA3   test passed!
HMAC-KDF    test passed!
GMAC     test passed!
```

```
Chacha   test passed!
POLY1305 test passed!
ChaCha20-Poly1305 AEAD test passed!
AES      test passed!
AES192   test passed!
AES256   test passed!
AES-GCM  test passed!
RSA      test passed!
DH       test passed!
PWDBASED test passed!
OPENSSL  test passed!
OPENSSL (EVP MD) passed!
OPENSSL (PKEY0) passed!
OPENSSL (PKEY1) passed!
OPENSSL (EVP Sign/Verify) passed!
ECC      test passed!
logging  test passed!
mutex    test passed!
memcb    test passed!
Test complete
Alternate cert chain used
 issuer : /C=US/ST=Montana/L=Bozeman/O=Sawtooth/OU=Consulting/CN=www.wolfssl.
    com/emailAddress=facts@wolfssl.com
 subject: /C=US/ST=Montana/L=Bozeman/O=wolfSSL/OU=Support/CN=www.wolfssl.com/
    emailAddress=facts@wolfssl.com
 altname = example.com
Alternate cert chain used
 issuer : /C=US/ST=Montana/L=Bozeman/O=wolfSSL_2048/OU=Programming-2048/CN=www
    .wolfssl.com/emailAddress=facts@wolfssl.com
 subject: /C=US/ST=Montana/L=Bozeman/O=wolfSSL_2048/OU=Programming-2048/CN=www
    .wolfssl.com/emailAddress=facts@wolfssl.com
 altname = example.com
 serial number:01
SSL version is TLSv1.2
SSL cipher suite is TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
SSL signature algorithm is RSA-SHA256
SSL curve name is SECP256R1
Session timeout set to 500 seconds
Client Random :  serial number:f1:5c:99:43:66:3d:96:04
SSL version is TLSv1.2
SSL cipher suite is TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
SSL signature algorithm is RSA-SHA256
SSL curve name is SECP256R1
1DC16A2C0D3AC49FC221DD5B8346B7B38CB9899B7A402341482183Server Random : 1679
   E50DBBBB3DB88C90F600C4C578F4F5D3CEAEC9B16BCCA215C276B448
765A1385611D6A
Client message: hello wolfssl!
I hear you fa shizzle!
sending server shutdown command: quit!
client sent quit command: shutting down!
ciphers = TLS13-AES128-GCM-SHA256:TLS13-AES256-GCM-SHA384:TLS13-CHACHA20-
   POLY1305-SHA256:DHE-RSA-AES128-SHA:DHE-RSA-AES256-SHA:ECDHE-RSA-AES128-SHA:
   ECDHE-RSA-AES256-SHA:ECDHE-ECDSA-AES128-SHA:ECDHE-ECDSA-AES256-SHA:DHE-RSA-
   AES128-SHA256:DHE-RSA-AES256-SHA256:DHE-RSA-AES128-GCM-SHA256:DHE-RSA-
```

```
    AES256-GCM-SHA384:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-RSA-AES256-GCM-SHA384:
    ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-
    AES128-SHA256:ECDHE-ECDSA-AES128-SHA256:ECDHE-RSA-AES256-SHA384:ECDHE-ECDSA
    -AES256-SHA384:ECDHE-RSA-CHACHA20-POLY1305:ECDHE-ECDSA-CHACHA20-POLY1305:
    DHE-RSA-CHACHA20-POLY1305:ECDHE-RSA-CHACHA20-POLY1305-OLD:ECDHE-ECDSA-
    CHACHA20-POLY1305-OLD:DHE-RSA-CHACHA20-POLY1305-OLD
33bc1a4570f4f1abccd5c48aace529b01a42ab51293954a297796e90d20970f0  input
33bc1a4570f4f1abccd5c48aace529b01a42ab51293954a297796e90d20970f0  /tmp/output-
    gNQWZL

All tests passed!
```

This indicates that everything is configured and built correctly. If any of the tests fail, make sure the build system was set up correctly. Likely culprits include having the wrong endianness or not properly setting the 64-bit type. If you've set anything to the non-default settings try removing those, rebuilding wolfSSL, and then re-testing.

3.3 Client Example

You can use the client example found in examples/client to test wolfSSL against any SSL server. To see a list of available command line runtime options, run the client with the `--help` argument:

```
./examples/client/client --help
```

Which returns:

```
wolfSSL client 4.8.1 NOTE: All files relative to wolfSSL home dir
Max RSA key size in bits for build is set at : 4096
-? <num>    Help, print this usage
            0: English, 1: Japanese
--help      Help, in English
-h <host>   Host to connect to, default 127.0.0.1
-p <num>    Port to connect on, not 0, default 11111
-v <num>    SSL version [0-4], SSLv3(0) - TLS1.3(4)), default 3
-V          Prints valid ssl version numbers, SSLv3(0) - TLS1.3(4)
-l <str>    Cipher suite list (: delimited)
-c <file>   Certificate file,           default ./certs/client-cert.pem
-k <file>   Key file,                   default ./certs/client-key.pem
-A <file>   Certificate Authority file, default ./certs/ca-cert.pem
-Z <num>    Minimum DH key bits,        default 1024
-b <num>    Benchmark <num> connections and print stats
-B <num>    Benchmark throughput using <num> bytes and print stats
-d          Disable peer checks
-D          Override Date Errors example
-e          List Every cipher suite available,
-g          Send server HTTP GET
-u          Use UDP DTLS, add -v 2 for DTLSv1, -v 3 for DTLSv1.2 (default)
-m          Match domain name in cert
-N          Use Non-blocking sockets
-r          Resume session
-w          Wait for bidirectional shutdown
-M <prot>   Use STARTTLS, using <prot> protocol (smtp)
-f          Fewer packet/group messages
-x          Disable client cert/key loading
-X          Driven by eXternal test case
-j          Use verify callback override
```

```
-n          Disable Extended Master Secret
-H <arg>    Internal tests [defCipherList, exitWithRet, verifyFail,
   useSupCurve,
                           loadSSL, disallowETM]
-J          Use HelloRetryRequest to choose group for KE
-K          Key Exchange for PSK not using (EC)DHE
-I          Update keys and IVs before sending data
-y          Key Share with FFDHE named groups only
-Y          Key Share with ECC named groups only
-1 <num>    Display a result by specified language.
            0: English, 1: Japanese
-2          Disable DH Prime check
-6          Simulate WANT_WRITE errors on every other IO send
-7          Set minimum downgrade protocol version [0-4]  SSLv3(0) - TLS1.3(4)
```

To test against example.com:443 try the following. This is using wolfSSL compiled with the `--enable-opensslextra` and `--enable-supportedcurves` build options:

```
./examples/client/client -h example.com -p 443 -d -g
```

Which returns:

```
Alternate cert chain used
 issuer : /C=US/O=DigiCert Inc/CN=DigiCert TLS RSA SHA256 2020 CA1
 subject: /C=US/ST=California/L=Los Angeles/O=Internet Corporation for
    Assigned Names and Numbers/CN=www.example.org
 altname = www.example.net
 altname = www.example.edu
 altname = www.example.com
 altname = example.org
 altname = example.net
 altname = example.edu
 altname = example.com
 altname = www.example.org
 serial number:0f:be:08:b0:85:4d:05:73:8a:b0:cc:e1:c9:af:ee:c9
SSL version is TLSv1.2
SSL cipher suite is TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
SSL curve name is SECP256R1
Session timeout set to 500 seconds
Client Random : 20640
   B8131D8E542646D395B362354F9308057B1624C2442C0B5FCDD064BFE29
SSL connect ok, sending GET...
HTTP/1.0 200 OK
Accept-Ranges: bytes
Content-Type: text/html
Date: Thu, 14 Oct 2021 16:50:28 GMT
Last-Modified: Thu, 14 Oct 2021 16:45:10 GMT
Server: ECS (nyb/1D10)
Content-Length: 94
Connection: close
```

This tells the client to connect to (`-h`) example.com on the HTTPS port (`-p`) of 443 and sends a generic (`-g`) GET request. The (`-d`) option tells the client not to verify the server. The rest is the initial output from the server that fits into the read buffer.

If no command line arguments are given, then the client attempts to connect to the localhost on the

wolfSSL default port of 11111. It also loads the client certificate in case the server wants to perform client authentication.

The client is able to benchmark a connection when using the `-b <num>` argument. When used, the client attempts to connect to the specified server/port the argument number of times and gives the average time in milliseconds that it took to perform `SSL_connect()`. For example:

```
./examples/client/client -b 100 -h example.com -p 443 -d
```

Returns:

```
wolfSSL_connect avg took:  296.417 milliseconds
```

If you'd like to change the default host from localhost, or the default port from 11111, you can change these settings in `wolfssl/test.h`. The variables wolfSSLIP and wolfSSLPort control these settings. Re-build all of the examples including testsuite when changing these settings otherwise the test programs won't be able to connect to each other.

By default, the wolfSSL example client tries to connect to the specified server using TLS 1.2. The user is able to change the SSL/TLS version which the client uses by passing the `-v` command line option. The following values are available for this option:

- `-v 0` - SSL 3.0 (disabled by default)
- `-v 1` - TLS 1.0
- `-v 2` - TLS 1.1
- `-v 3` - TLS 1.2 (selected by default)
- `-v 4` - TLS 1.3

A common error users see when using the example client is -188:

```
wolfSSL_connect error -188, ASN no signer error to confirm failure
wolfSSL error: wolfSSL_connect failed
```

This is typically caused by the wolfSSL client not being able to verify the certificate of the server it is connecting to. By default, the wolfSSL client loads the yaSSL test CA certificate as a trusted root certificate. This test CA certificate will not be able to verify an external server certificate which was signed by a different CA. As such, to solve this problem, users either need to turn off verification of the peer (server), using the `-d` option:

```
./examples/client/client -h myhost.com -p 443 -d
```

Or load the correct CA certificate into the wolfSSL client using the `-A` command line option:

```
./examples/client/client -h myhost.com -p 443 -A serverCA.pem
```

3.4 Server Example

The server example demonstrates a simple SSL server that optionally performs client authentication. Only one client connection is accepted and then the server quits. The client example in normal mode (no command line arguments) will work just fine against the example server, but if you specify command line arguments for the client example, then a client certificate isn't loaded and the `wolfSSL_connect()` will fail (unless client cert check is disabled using the `-d` option). The server will report an error "-245, peer didn't send cert". Like the example client, the server can be used with several command line arguments as well:

```
./examples/server/server --help
```

Which returns:

```
server 4.8.1 NOTE: All files relative to wolfSSL home dir
-? <num>    Help, print this usage
```

```
            0: English, 1: Japanese
--help      Help, in English
-p <num>    Port to listen on, not 0, default 11111
-v <num>    SSL version [0-4], SSLv3(0) - TLS1.3(4)), default 3
-l <str>    Cipher suite list (: delimited)
-c <file>   Certificate file,           default ./certs/server-cert.pem
-k <file>   Key file,                   default ./certs/server-key.pem
-A <file>   Certificate Authority file, default ./certs/client-cert.pem
-R <file>   Create Ready file for external monitor default none
-D <file>   Diffie-Hellman Params file, default ./certs/dh2048.pem
-Z <num>    Minimum DH key bits,        default 1024
-d          Disable client cert check
-b          Bind to any interface instead of localhost only
-s          Use pre Shared keys
-u          Use UDP DTLS, add -v 2 for DTLSv1, -v 3 for DTLSv1.2 (default)
-f          Fewer packet/group messages
-r          Allow one client Resumption
-N          Use Non-blocking sockets
-S <str>    Use Host Name Indication
-w          Wait for bidirectional shutdown
-x          Print server errors but do not close connection
-i          Loop indefinitely (allow repeated connections)
-e          Echo data mode (return raw bytes received)
-B <num>    Benchmark throughput using <num> bytes and print stats
-g          Return basic HTML web page
-C <num>    The number of connections to accept, default: 1
-H <arg>    Internal tests [defCipherList, exitWithRet, verifyFail,
   useSupCurve,
                           loadSSL, disallowETM]
-U          Update keys and IVs before sending
-K          Key Exchange for PSK not using (EC)DHE
-y          Pre-generate Key Share using FFDHE_2048 only
-Y          Pre-generate Key Share using P-256 only
-F          Send alert if no mutual authentication
-2          Disable DH Prime check
-1 <num>    Display a result by specified language.
            0: English, 1: Japanese
-6          Simulate WANT_WRITE errors on every other IO send
-7          Set minimum downgrade protocol version [0-4]  SSLv3(0) - TLS1.3(4)
```

3.5 EchoServer Example

The echoserver example sits in an endless loop waiting for an unlimited number of client connections. Whatever the client sends the echoserver echoes back. Client authentication isn't performed so the example client can be used against the echoserver in all 3 modes. Four special commands aren't echoed back and instruct the echoserver to take a different action.

1. `quit` - If the echoserver receives the string "quit" it will shutdown.
2. `break` If the echoserver receives the string "break" it will stop the current session but continue handling requests. This is particularly useful for DTLS testing.
3. `printstats` - If the echoserver receives the string "printstats" it will print out statistics for the session cache.
4. `GET` - If the echoserver receives the string "GET" it will handle it as an http get and send back a simple page with the message "greeting from wolfSSL". This allows testing of various TLS/SSL

clients like Safari, IE, Firefox, gnutls, and the like against the echoserver example.

The output of the echoserver is echoed to stdout unless NO_MAIN_DRIVER is defined. You can redirect output through the shell or through the first command line argument. To create a file named output.txt with the output from the echoserver run:

```
./examples/echoserver/echoserver output.txt
```

3.6 EchoClient Example

The echoclient example can be run in interactive mode or batch mode with files. To run in interactive mode and write 3 strings "hello", "wolfssl", and "quit" results in:

```
./examples/echoclient/echoclient
hello
hello
wolfssl
wolfssl
quit
sending server shutdown command: quit!
```

To use an input file, specify the filename on the command line as the first argument. To echo the contents of the file input.txt issue:

```
./examples/echoclient/echoclient input.txt
```

If you want the result to be written out to a file, you can specify the output file name as an additional command line argument. The following command will echo the contents of file input.txt and write the result from the server to output.txt:

```
./examples/echoclient/echoclient input.txt output.txt
```

The testsuite program does just that, but hashes the input and output files to make sure that the client and server were getting/sending the correct and expected results.

3.7 Benchmark

Many users are curious about how the wolfSSL embedded SSL library will perform on a specific hardware device or in a specific environment. Because of the wide variety of different platforms and compilers used today in embedded, enterprise, and cloud-based environments, it is hard to give generic performance calculations across the board.

To help wolfSSL users and customers in determining SSL performance for wolfSSL / wolfCrypt, a benchmark application is provided which is bundled with wolfSSL. wolfSSL uses the wolfCrypt cryptography library for all crypto operations by default. Because the underlying crypto is a very performance-critical aspect of SSL/TLS, our benchmark application runs performance tests on wolfCrypt's algorithms.

The benchmark utility located in wolfcrypt/benchmark (./wolfcrypt/benchmark/benchmark) may be used to benchmark the cryptographic functionality of wolfCrypt. Typical output may look like the following (in this output, several optional algorithms/ciphers were enabled including ECC, SHA-256, SHA-512, AES-GCM, AES-CCM, and Camellia):

```
./wolfcrypt/benchmark/benchmark

------------------------------------------------------------------------------
 wolfSSL version 4.8.1
------------------------------------------------------------------------------
wolfCrypt Benchmark (block bytes 1048576, min 1.0 sec each)
```

```
RNG                105 MB took 1.004 seconds,  104.576 MB/s Cycles per byte =
     20.94
AES-128-CBC-enc    310 MB took 1.008 seconds,  307.434 MB/s Cycles per byte =
      7.12
AES-128-CBC-dec    290 MB took 1.002 seconds,  289.461 MB/s Cycles per byte =
      7.56
AES-192-CBC-enc    265 MB took 1.010 seconds,  262.272 MB/s Cycles per byte =
      8.35
AES-192-CBC-dec    240 MB took 1.013 seconds,  236.844 MB/s Cycles per byte =
      9.24
AES-256-CBC-enc    240 MB took 1.011 seconds,  237.340 MB/s Cycles per byte =
      9.22
AES-256-CBC-dec    235 MB took 1.018 seconds,  230.864 MB/s Cycles per byte =
      9.48
AES-128-GCM-enc    160 MB took 1.011 seconds,  158.253 MB/s Cycles per byte =
     13.83
AES-128-GCM-dec    160 MB took 1.016 seconds,  157.508 MB/s Cycles per byte =
     13.90
AES-192-GCM-enc    150 MB took 1.022 seconds,  146.815 MB/s Cycles per byte =
     14.91
AES-192-GCM-dec    150 MB took 1.039 seconds,  144.419 MB/s Cycles per byte =
     15.16
AES-256-GCM-enc    130 MB took 1.017 seconds,  127.889 MB/s Cycles per byte =
     17.12
AES-256-GCM-dec    140 MB took 1.030 seconds,  135.943 MB/s Cycles per byte =
     16.10
GMAC Table 4-bit   321 MB took 1.002 seconds,  320.457 MB/s Cycles per byte =
      6.83
CHACHA             420 MB took 1.002 seconds,  419.252 MB/s Cycles per byte =
      5.22
CHA-POLY           330 MB took 1.013 seconds,  325.735 MB/s Cycles per byte =
      6.72
MD5                655 MB took 1.007 seconds,  650.701 MB/s Cycles per byte =
      3.36
POLY1305          1490 MB took 1.002 seconds, 1486.840 MB/s Cycles per byte =
      1.47
SHA                560 MB took 1.004 seconds,  557.620 MB/s Cycles per byte =
      3.93
SHA-224            240 MB took 1.011 seconds,  237.474 MB/s Cycles per byte =
      9.22
SHA-256            250 MB took 1.020 seconds,  245.081 MB/s Cycles per byte =
      8.93
SHA-384            380 MB took 1.005 seconds,  377.963 MB/s Cycles per byte =
      5.79
SHA-512            380 MB took 1.007 seconds,  377.260 MB/s Cycles per byte =
      5.80
SHA3-224           385 MB took 1.009 seconds,  381.679 MB/s Cycles per byte =
      5.74
SHA3-256           360 MB took 1.004 seconds,  358.583 MB/s Cycles per byte =
      6.11
SHA3-384           270 MB took 1.020 seconds,  264.606 MB/s Cycles per byte =
      8.27
SHA3-512           185 MB took 1.019 seconds,  181.573 MB/s Cycles per byte =
     12.06
```

```
HMAC-MD5           665 MB took 1.004 seconds,  662.154 MB/s Cycles per byte =
     3.31
HMAC-SHA           590 MB took 1.004 seconds,  587.535 MB/s Cycles per byte =
     3.73
HMAC-SHA224        240 MB took 1.018 seconds,  235.850 MB/s Cycles per byte =
     9.28
HMAC-SHA256        245 MB took 1.013 seconds,  241.805 MB/s Cycles per byte =
     9.05
HMAC-SHA384        365 MB took 1.006 seconds,  362.678 MB/s Cycles per byte =
     6.04
HMAC-SHA512        365 MB took 1.009 seconds,  361.674 MB/s Cycles per byte =
     6.05
PBKDF2              30 KB took 1.000 seconds,   29.956 KB/s Cycles per byte =
    74838.56
RSA     2048 public      18400 ops took 1.004 sec, avg 0.055 ms, 18335.019 ops
    /sec
RSA     2048 private       300 ops took 1.215 sec, avg 4.050 ms, 246.891 ops/
    sec
DH      2048 key gen      1746 ops took 1.000 sec, avg 0.573 ms, 1745.991 ops/
    sec
DH      2048 agree         900 ops took 1.060 sec, avg 1.178 ms, 849.210 ops/
    sec
ECC   [      SECP256R1]   256 key gen       901 ops took 1.000 sec, avg 1.110
    ms, 900.779 ops/sec
ECDHE [      SECP256R1]   256 agree        1000 ops took 1.105 sec, avg 1.105
    ms, 904.767 ops/sec
ECDSA [      SECP256R1]   256 sign          900 ops took 1.022 sec, avg 1.135
    ms, 880.674 ops/sec
ECDSA [      SECP256R1]   256 verify       1300 ops took 1.012 sec, avg 0.779
    ms, 1284.509 ops/sec
Benchmark complete
```

This is especially useful for comparing the public key speed before and after changing the math library. You can test the results using the normal math library (`./configure`), the fastmath library (`./configure --enable-fastmath`), and the fasthugemath library (`./configure --enable-fasthugemath`).

For more details and benchmark results, please refer to the wolfSSL Benchmarks page: https://www.wolfssl.com/docs/benchmarks

3.7.1 Relative Performance

Although the performance of individual ciphers and algorithms will depend on the host platform, the following graph shows relative performance between wolfCrypt's ciphers. These tests were conducted on a Macbook Pro (OS X 10.6.8) running a 2.2 GHz Intel Core i7.

If you want to use only a subset of ciphers, you can customize which specific cipher suites and/or ciphers wolfSSL uses when making an SSL/TLS connection. For example, to force 128-bit AES, add the following line after the call to `wolfSSL_CTX_new(SSL_CTX_new)`:

```
wolfSSL_CTX_set_cipher_list(ctx, "AES128-SHA");
```

3.7.2 Benchmarking Notes

1. The processors native register size (32 vs 64-bit) can make a big difference when doing 1000+ bit public key operations.

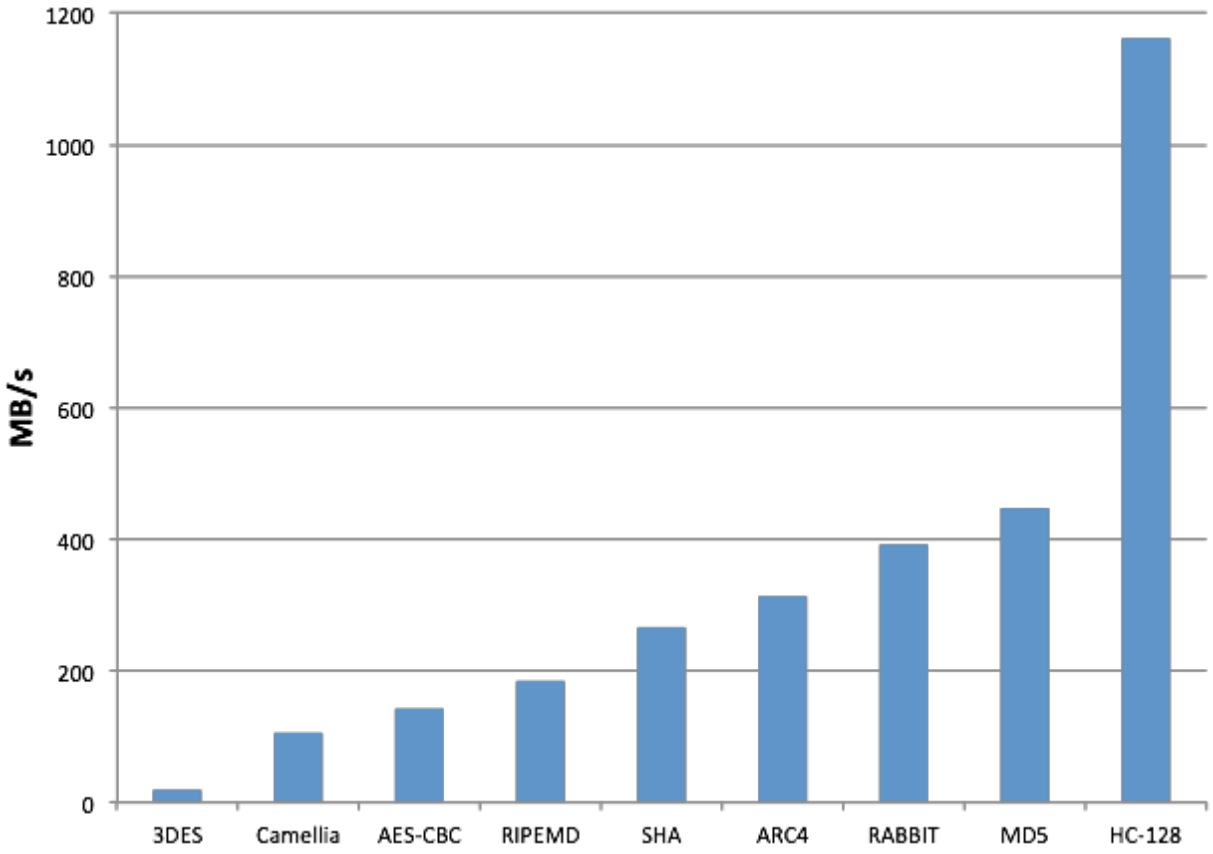

Figure 1: Benchmark

2. **keygen** (`--enable-keygen`) will allow you to also benchmark key generation speeds when running the benchmark utility.
3. **fastmath** (`--enable-fastmath`) reduces dynamic memory usage and speeds up public key operations. If you are having trouble building on 32-bit platform with fastmath, disable shared libraries so that PIC isn't hogging a register (also see notes in the README):

   ```
   ./configure --enable-fastmath --disable-shared
   make clean
   make
   ```

   Note: doing a `make clean` is good practice with wolfSSL when switching configure options.
4. By default, fastmath tries to use assembly optimizations if possible. If assembly optimizations don't work, you can still use fastmath without them by adding `TFM_NO_ASM` to `CFLAGS` when building wolfSSL:

   ```
   ./configure --enable-fastmath C_EXTRA_FLAGS="-DTFM_NO_ASM"
   ```
5. Using fasthugemath can try to push fastmath even more for users who are not running on embedded platforms:

   ```
   ./configure --enable-fasthugemath
   ```
6. With the default wolfSSL build, we have tried to find a good balance between memory usage and performance. If you are more concerned about one of the two, please refer back to Build Options for additional wolfSSL configuration options.
7. **Bulk Transfers**: wolfSSL by default uses 128 byte I/O buffers since about 80% of SSL traffic falls within this size and to limit dynamic memory use. It can be configured to use 16K buffers (the maximum SSL size) if bulk transfers are required.

3.7.3 Benchmarking on Embedded Systems

There are several build options available to make building the benchmark application on an embedded system easier. These include:

3.7.3.1 BENCH_EMBEDDED Enabling this define will switch the benchmark application from using Megabytes to using Kilobytes, therefore reducing the memory usage. By default, when using this define, ciphers and algorithms will be benchmarked with 25kB. Public key algorithms will only be benchmarked over 1 iteration (as public key operations on some embedded processors can be fairly slow). These can be adjusted in `benchmark.c` by altering the variables `numBlocks` and `ntimes` located inside the `BENCH_EMBEDDED` define.

3.7.3.2 USE_CERT_BUFFERS_1024 Enabling this define will switch the benchmark application from loading test keys and certificates from the file system and instead use 1024-bit key and certificate buffers located in `<wolfssl_root>/wolfssl/certs_test.h`. It is useful to use this define when an embedded platform has no filesystem (used with `NO_FILESYSTEM`) and a slow processor where 2048-bit public key operations may not be reasonable.

3.7.3.3 USE_CERT_BUFFERS_2048 Enabling this define is similar to `USE_CERT_BUFFERS_1024` except that 2048-bit key and certificate buffers are used instead of 1024-bit ones. This define is useful when the processor is fast enough to do 2048-bit public key operations but when there is no filesystem available to load keys and certificates from files.

3.8 Changing a Client Application to Use wolfSSL

This section will explain the basic steps needed to add wolfSSL to a client application, using the wolfSSL native API. For a server explanation, please see Changing a Server Application to Use wolfSSL. A more complete walk-through with example code is located in the SSL Tutorial in Chapter 11. If you want more information about the OpenSSL compatibility layer, please see OpenSSL Compatibility.

1. Include the wolfSSL header:

   ```
   #include <wolfssl/ssl.h>
   ```

2. Initialize wolfSSL and the WOLFSSL_CTX. You can use one WOLFSSL_CTX no matter how many WOLFSSL objects you end up creating. You'll just need to load CA certificates to verify the server you are connecting to. Basic initialization looks like:

   ```
   wolfSSL_Init();
   WOLFSSL_CTX* ctx;
   if ((ctx = wolfSSL_CTX_new(wolfTLSv1_client_method())) == NULL)
   {
       fprintf(stderr, "wolfSSL_CTX_new error.\n");
       exit(EXIT_FAILURE);
   }
   if (wolfSSL_CTX_load_verify_locations(ctx,"./ca-cert.pem",0) !=
    ↪  SSL_SUCCESS) {
       fprintf(stderr, "Error loading ./ca-cert.pem,"
                       " please check the file.\n");
       exit(EXIT_FAILURE);
   }
   ```

3. Create the WOLFSSL object after each TCP connect and associate the file descriptor with the session:

   ```
   /*after connecting to socket fd*/
   WOLFSSL* ssl;
   if ((ssl = wolfSSL_new(ctx)) == NULL) {
       fprintf(stderr, "wolfSSL_new error.\n");
       exit(EXIT_FAILURE);
   }
   wolfSSL_set_fd(ssl, fd);
   ```

4. Change all calls from read() (or recv()) to wolfSSL_read() so:

   ```
   result = read(fd, buffer, bytes);
   ```

   becomes:

   ```
   result = wolfSSL_read(ssl, buffer, bytes);
   ```

5. Change all calls from write() (or send()) to wolfSSL_write() so:

   ```
   result = write(fd, buffer, bytes);
   ```

   becomes

   ```
   result = wolfSSL_write(ssl, buffer, bytes);
   ```

6. You can manually call wolfSSL_connect() but that's not even necessary; the first call to wolfSSL_read() or wolfSSL_write() will initiate the wolfSSL_connect() if it hasn't taken place yet.

7. Error checking. Each wolfSSL_read() and wolfSSL_write() call will return the number of bytes written upon success, 0 upon connection closure, and -1 for an error, just like read() and

write(). In the event of an error you can use two calls to get more information about the error:

```
char errorString[80];
int err = wolfSSL_get_error(ssl, 0);
wolfSSL_ERR_error_string(err, errorString);
```

If you are using non-blocking sockets, you can test for errno EAGAIN/EWOULDBLOCK or more correctly you can test the specific error code returned by wolfSSL_get_error() for SSL_ERROR_WANT_READ orSSL_ERROR_WANT_WRITE.

8. Cleanup. After each WOLFSSL object is done being used you can free it up by calling:

   ```
   wolfSSL_free(ssl);
   ```

   When you are completely done using SSL/TLS altogether you can free the WOLFSSL_CTX object by calling:

   ```
   wolfSSL_CTX_free(ctx);
   wolfSSL_Cleanup();
   ```

   For an example of a client application using wolfSSL, see the client example located in the <wolfssl_root>/examples/client/client.c file.

3.9 Changing a Server Application to Use wolfSSL

This section will explain the basic steps needed to add wolfSSL to a server application using the wolfSSL native API. For a client explanation, please see Changing a Client Application to Use wolfSSL. A more complete walk-through, with example code, is located in the SSL Tutorial chapter.

1. Follow the instructions above for a client, except change the client method call in step 2 to a server one, so:

   ```
   wolfSSL_CTX_new(wolfTLSv1_client_method());
   ```

   becomes:

   ```
   wolfSSL_CTX_new(wolfTLSv1_server_method());
   ```

   or even:

   ```
   wolfSSL_CTX_new(wolfSSLv23_server_method());
   ```

   To allow SSLv3 and TLSv1+ clients to connect to the server.

2. Add the server's certificate and key file to the initialization in step 5 above:

   ```
   if (wolfSSL_CTX_use_certificate_file(ctx,"./server-cert.pem",
    ↪  SSL_FILETYPE_PEM) != SSL_SUCCESS) {
       fprintf(stderr, "Error loading ./server-cert.pem,"
                       " please check the file.\n");
       exit(EXIT_FAILURE);
   }
   if (wolfSSL_CTX_use_PrivateKey_file(ctx,"./server-key.pem",
    ↪  SSL_FILETYPE_PEM) != SSL_SUCCESS) {
       fprintf(stderr, "Error loading ./server-key.pem,"
                       " please check the file.\n");
       exit(EXIT_FAILURE);
   }
   ```

It is possible to load certificates and keys from buffers as well if there is no filesystem available. In this case, see the wolfSSL_CTX_use_certificate_buffer() and wolfSSL_CTX_use_PrivateKey_buffer() API documentation, linked here, for more information.

For an example of a server application using wolfSSL, see the server example located in the <wolfssl_root>/examples/server/server.c file.

4 Features

wolfSSL (formerly CyaSSL) supports the C programming language as a primary interface, but also supports several other host languages, including Java, PHP, Perl, and Python (through a SWIG interface). If you have interest in hosting wolfSSL in another programming language that is not currently supported, please contact us.

This chapter covers some of the features of wolfSSL in more depth, including Stream Ciphers, AES-NI, IPv6 support, SSL Inspection (Sniffer) support, and more.

4.1 Features Overview

For an overview of wolfSSL features, please reference the wolfSSL product webpage: https://www.wolfssl.com/products/wolfssl

4.2 Protocol Support

wolfSSL supports **SSL 3.0**, **TLS** (**1.0**, **1.1**, **1.2, 1.3**), and **DTLS** (**1.0**, **1.2**, **1.3**). You can easily select a protocol to use by using one of the following functions (as shown for either the client or server). wolfSSL does not support SSL 2.0, as it has been insecure for several years. The client and server functions below change slightly when using the OpenSSL compatibility layer. For the OpenSSL-compatible functions, please see OpenSSL Compatibility.

4.2.1 Server Functions

- `wolfDTLSv1_server_method()` - DTLS 1.0
- `wolfDTLSv1_2_server_method()` - DTLS 1.2
- `wolfSSLv3_server_method()` - SSL 3.0
- `wolfTLSv1_server_method()` - TLS 1.0
- `wolfTLSv1_1_server_method()` - TLS 1.1
- `wolfTLSv1_2_server_method()` - TLS 1.2
- `wolfTLSv1_3_server_method()` - TLS 1.3
- `wolfSSLv23_server_method()` - Use highest possible version from SSLv3 - TLS 1.2

wolfSSL supports robust server downgrade with the `wolfSSLv23_server_method()` function. See Robust Client and Server Downgrade for details.

4.2.2 Client Functions

- `wolfDTLSv1_client_method()` - DTLS 1.0
- `wolfDTLSv1_2_client_method_ex()` - DTLS 1.2
- `wolfSSLv3_client_method()` - SSL 3.0
- `wolfTLSv1_client_method()` - TLS 1.0
- `wolfTLSv1_1_client_method()` - TLS 1.1
- `wolfTLSv1_2_client_method()` - TLS 1.2
- `wolfTLSv1_3_client_method()` - TLS 1.3
- `wolfSSLv23_client_method()` - Use highest possible version from SSLv3 - TLS 1.2

wolfSSL supports robust client downgrade with the `wolfSSLv23_client_method()` function. See Robust Client and Server Downgrade for details.

For details on how to use these functions, please see the Getting Started chapter. For a comparison between SSL, TLS, and DTLS, please see Appendix A.

COPYRIGHT ©2024 wolfSSL Inc.

4.2.3 Robust Client and Server Downgrade

Both wolfSSL clients and servers have robust version downgrade capability. If a specific protocol version method is used on either side, then only that protocol version will be negotiated or an error will be returned. For example, if a client that uses TLS 1.0 tries to connect to an SSL 3.0 only server, then the connection will fail; likewise, connecting to a TLS 1.1 server will fail as well.

To resolve this issue, a client that uses the `wolfSSLv23_client_method()` function will support the highest protocol version supported by the server by downgrading if necessary. In this case, the client will be able to connect to a server running TLS 1.0 - TLS 1.3 (or a subset or superset that includes SSL 3.0 depending on which protocol versions are configured in wolfSSL). The only versions it can't connect to are SSL 2.0 which has been insecure for years, and SSL 3.0 which has been disabled by default.

Similarly, a server using the `wolfSSLv23_server_method()` function can handle clients supporting protocol versions from TLS 1.0 - TLS 1.3. A wolfSSL server can't accept a connection from SSLv2 because no security is provided.

4.2.4 IPv6 Support

If you are an adopter of IPv6 and want to use an embedded SSL implementation then you may have been wondering if wolfSSL supports IPv6. The answer is yes, we do support wolfSSL running on top of IPv6.

wolfSSL was designed as IP neutral, and will work with both IPv4 and IPv6, but the current test applications default to IPv4 (so as to apply to a broader range of systems). To change the test applications to IPv6, use the **–enable-ipv6** option while building wolfSSL.

Further information on IPv6 can be found here:

https://en.wikipedia.org/wiki/IPv6.

4.2.5 DTLS

wolfSSL has support for DTLS ("Datagram" TLS) for both client and server. The current supported versions are DTLS 1.0, 1.2, and 1.3.

The TLS protocol was designed to provide a secure transport channel across a **reliable** medium (such as TCP). As application layer protocols began to be developed using UDP transport (such as SIP and various electronic gaming protocols), a need arose for a way to provide communications security for applications which are delay sensitive. This need led to the creation of the DTLS protocol.

Many people believe the difference between TLS and DTLS is the same as TCP vs. UDP. This is incorrect. UDP has the benefit of having no handshake, no tear-down, and no delay in the middle if something gets lost (compared with TCP). DTLS on the other hand, has an extended SSL handshake and tear-down and must implement TCP-like behavior for the handshake. In essence, DTLS reverses the benefits that are offered by UDP in exchange for a secure connection.

DTLS can be enabled when building wolfSSL by using the `--enable-dtls` build option.

4.2.6 LwIP (Lightweight Internet Protocol)

wolfSSL supports the lightweight internet protocol implementation out of the box. To use this protocol all you need to do is define `WOLFSSL_LWIP` or navigate to the `settings.h` file and uncomment the line:

```
/*#define WOLFSSL_LWIP*/
```

The focus of lwIP is to reduce RAM usage while still providing a full TCP stack. That focus makes lwIP great for use in embedded systems, an area where wolfSSL is an ideal match for SSL/TLS needs.

4.2.7 TLS Extensions

A list of TLS extensions supported by wolfSSL and the RFC corresponding to each extension:

RFC	Extension	wolfSSL Type
6066	Server Name Indication	TLSX_SERVER_NAME
6066	Maximum Fragment Length Negotiation	TLSX_MAX_FRAGMENT_LENGTH
6066	Truncated HMAC	TLSX_TRUNCATED_HMAC
6066	Status Request	TLSX_STATUS_REQUEST
7919	Supported Groups	TLSX_SUPPORTED_GROUPS
5246	Signature Algorithm	TLSX_SIGNATURE_ALGORITHMS
7301	Application Layer Protocol Negotiation	TLSX_APPLICATION_LAYER_PROTOCOL
6961	Multiple Certificate Status Request	TLSX_STATUS_REQUEST_V2
5077	Session Ticket	TLSX_SESSION_TICKET
5746	Renegotiation Indication	TLSX_RENEGOTIATION_INFO
8446	Key Share	TLSX_KEY_SHARE
8446	Pre Shared Key	TLSX_PRE_SHARED_KEY
8446	PSK Key Exchange Modes	TLSX_PSK_KEY_EXCHANGE_MODES
8446	Early Data	TLSX_EARLY_DATA
8446	Cookie	TLSX_COOKIE
8446	Supported Versions	TLSX_SUPPORTED_VERSIONS
8446	Post Handshake Authorization	TLSX_POST_HANDSHAKE_AUTH

4.3 Cipher Support

4.3.1 Cipher Suite Strength and Choosing Proper Key Sizes

To see what ciphers are currently being used you can call the method: `wolfSSL_get_ciphers()`.

This function will return the currently enabled cipher suites.

Cipher suites come in a variety of strengths. Because they are made up of several different types of algorithms (authentication, encryption, and message authentication code (MAC)), the strength of each varies with the chosen key sizes.

There can be many methods of grading the strength of a cipher suite - the specific method used seems to vary between different projects and companies and can include things such as symmetric and public key algorithm key sizes, type of algorithm, performance, and known weaknesses.

NIST (National Institute of Standards and Technology) makes recommendations on choosing an acceptable cipher suite by providing comparable algorithm strengths for varying key sizes of each. The NIST Special Publication, SP800-57, states how two algorithms can be compared:

> Two algorithms are considered to be of comparable strength for the given key sizes (X and Y) if the amount of work needed to "break the algorithms" or determine the keys (with the given key sizes) is approximately the same using a given resource. The security strength of an algorithm for a given key size is traditionally described in terms of the amount of work it takes to try all keys for a symmetric algorithm with a key size of "X" that has no shortcut attacks (i.e., the most efficient attack is to try all possible keys).

The following two tables are based off of both Table 2 (pg. 56) and Table 4 (pg. 59) from NIST SP800-57, and shows comparable security strength between algorithms as well as a strength measurement (based off of NIST's suggested algorithm security lifetimes using bits of security).

Note: In the following table "L" is the size of the public key for finite field cryptography (FFC), "N" is the size of the private key for FFC, "k" is considered the key size for integer factorization cryptography (IFC), and "f" is considered the key size for elliptic curve cryptography.

Bits of Security	Symmetric Key Algorithms	**FFC** Key Size (DSA, DH, etc.)	**IFC** Key Size (RSA, etc.)	**ECC** Key Size (ECDSA, etc.)	Description
80	2TDEA, etc.	L = 1024, N = 160	k = 1024	f = 160-223	Security good through 2010
128	AES-128, etc.	L = 3072, N = 256	k = 3072	f = 256-383	Security good through 2030
192	AES-192, etc.	L = 7680, N = 384	k = 7680	f = 384-511	Long Term Protection
256	AES-256, etc.	L = 15360, N = 512	k = 15360	f = 512+	Secure for the fore-see-able future

Using this table as a guide, to begin to classify a cipher suite, we categorize it based on the strength of the symmetric encryption algorithm. In doing this, a rough grade classification can be devised to classify each cipher suite based on bits of security (only taking into account symmetric key size):

- **LOW** - bits of security smaller than 128 bits
- **MEDIUM** - bits of security equal to 128 bits
- **HIGH** - bits of security larger than 128 bits

Outside of the symmetric encryption algorithm strength, the strength of a cipher suite will depend greatly on the key sizes of the key exchange and authentication algorithm keys. The strength is only as good as the cipher suite's weakest link.

Following the above grading methodology (and only basing it on symmetric encryption algorithm strength), wolfSSL 2.0.0 currently supports a total of 0 LOW strength cipher suites, 12 MEDIUM strength cipher suites, and 8 HIGH strength cipher suites – as listed below. The following strength classification could change depending on the chosen key sizes of the other algorithms involved. For a reference on hash function security strength, see Table 3 (pg. 56) of NIST SP800-57.

In some cases, you will see ciphers referenced as "**EXPORT**" ciphers. These ciphers originated from the time period in US history (as late as 1992) when it was illegal to export software with strong encryption from the United States. Strong encryption was classified as "Munitions" by the US Government (under the same category as Nuclear Weapons, Tanks, and Ballistic Missiles). Because of this restriction, software being exported included "weakened" ciphers (mostly in smaller key sizes). In the current day, this restriction has been lifted, and as such, EXPORT ciphers are no longer a mandated necessity.

4.3.2 Supported Cipher Suites

The following cipher suites are supported by wolfSSL. A cipher suite is a combination of authentication, encryption, and message authentication code (MAC) algorithms which are used during the TLS or SSL

handshake to negotiate security settings for a connection.

Each cipher suite defines a key exchange algorithm, a bulk encryption algorithm, and a message authentication code algorithm (MAC). The **key exchange algorithm** (RSA, DSS, DH, EDH) determines how the client and server will authenticate during the handshake process. The **bulk encryption algorithm** (DES, 3DES, AES, ARC4), including block ciphers and stream ciphers, is used to encrypt the message stream. The **message authentication code (MAC) algorithm** (MD2, MD5, SHA-1, SHA-256, SHA-512, RIPEMD) is a hash function used to create the message digest.

The table below matches up to the cipher suites (and categories) found in `<wolfssl_root>/wolfssl/internal.h` (starting at about line 1097). If you are looking for a cipher suite which is not in the following list, please contact us to discuss getting it added to wolfSSL.

ECC cipher suites:

- TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA
- TLS_DHE_RSA_WITH_AES_256_CBC_SHA
- TLS_DHE_RSA_WITH_AES_128_CBC_SHA
- TLS_DH_anon_WITH_AES_128_CBC_SHA
- TLS_RSA_WITH_AES_256_CBC_SHA
- TLS_RSA_WITH_AES_128_CBC_SHA
- TLS_RSA_WITH_NULL_SHA
- TLS_PSK_WITH_AES_256_CBC_SHA
- TLS_PSK_WITH_AES_128_CBC_SHA256
- TLS_PSK_WITH_AES_256_CBC_SHA384
- TLS_PSK_WITH_AES_128_CBC_SHA
- TLS_PSK_WITH_NULL_SHA256
- TLS_PSK_WITH_NULL_SHA384
- TLS_PSK_WITH_NULL_SHA
- SSL_RSA_WITH_RC4_128_SHA
- SSL_RSA_WITH_RC4_128_MD5
- SSL_RSA_WITH_3DES_EDE_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_RSA_WITH_RC4_128_SHA
- TLS_ECDHE_ECDSA_WITH_RC4_128_SHA
- TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
- TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384
- TLS_ECDHE_PSK_WITH_NULL_SHA256
- TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_ECDSA_WITH_NULL_SHA

Static ECC cipher suites:

- TLS_ECDH_RSA_WITH_AES_256_CBC_SHA
- TLS_ECDH_RSA_WITH_AES_128_CBC_SHA
- TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA
- TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA
- TLS_ECDH_RSA_WITH_RC4_128_SHA
- TLS_ECDH_ECDSA_WITH_RC4_128_SHA
- TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA

- TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA
- TLS_ECDH_RSA_WITH_AES_128_CBC_SHA256
- TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256
- TLS_ECDH_RSA_WITH_AES_256_CBC_SHA384
- TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA384

Blake2b cipher suites:

- TLS_RSA_WITH_AES_128_CBC_B2B256
- TLS_RSA_WITH_AES_256_CBC_B2B256

SHA-256 cipher suites:

- TLS_DHE_RSA_WITH_AES_256_CBC_SHA256
- TLS_DHE_RSA_WITH_AES_128_CBC_SHA256
- TLS_RSA_WITH_AES_256_CBC_SHA256
- TLS_RSA_WITH_AES_128_CBC_SHA256
- TLS_RSA_WITH_NULL_SHA256
- TLS_DHE_PSK_WITH_AES_128_CBC_SHA256
- TLS_DHE_PSK_WITH_NULL_SHA256

SHA-384 cipher suites:

- TLS_DHE_PSK_WITH_AES_256_CBC_SHA384
- TLS_DHE_PSK_WITH_NULL_SHA384

AES-GCM cipher suites:

- TLS_RSA_WITH_AES_128_GCM_SHA256
- TLS_RSA_WITH_AES_256_GCM_SHA384
- TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_DHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_PSK_WITH_AES_128_GCM_SHA256
- TLS_PSK_WITH_AES_256_GCM_SHA384
- TLS_DHE_PSK_WITH_AES_128_GCM_SHA256
- TLS_DHE_PSK_WITH_AES_256_GCM_SHA384

ECC AES-GCM cipher suites:

- TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
- TLS_ECDH_ECDSA_WITH_AES_128_GCM_SHA256
- TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDH_RSA_WITH_AES_128_GCM_SHA256
- TLS_ECDH_RSA_WITH_AES_256_GCM_SHA384

AES-CCM cipher suites:

- TLS_RSA_WITH_AES_128_CCM_8
- TLS_RSA_WITH_AES_256_CCM_8
- TLS_ECDHE_ECDSA_WITH_AES_128_CCM
- TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8
- TLS_ECDHE_ECDSA_WITH_AES_256_CCM_8
- TLS_PSK_WITH_AES_128_CCM
- TLS_PSK_WITH_AES_256_CCM
- TLS_PSK_WITH_AES_128_CCM_8
- TLS_PSK_WITH_AES_256_CCM_8
- TLS_DHE_PSK_WITH_AES_128_CCM

- TLS_DHE_PSK_WITH_AES_256_CCM

Camellia cipher suites:

- TLS_RSA_WITH_CAMELLIA_128_CBC_SHA
- TLS_RSA_WITH_CAMELLIA_256_CBC_SHA
- TLS_RSA_WITH_CAMELLIA_128_CBC_SHA256
- TLS_RSA_WITH_CAMELLIA_256_CBC_SHA256
- TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA
- TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA
- TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA256
- TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA256

ChaCha cipher suites:

- TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256
- TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256
- TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256
- TLS_ECDHE_PSK_WITH_CHACHA20_POLY1305_SHA256
- TLS_PSK_WITH_CHACHA20_POLY1305_SHA256
- TLS_DHE_PSK_WITH_CHACHA20_POLY1305_SHA256
- TLS_ECDHE_RSA_WITH_CHACHA20_OLD_POLY1305_SHA256
- TLS_ECDHE_ECDSA_WITH_CHACHA20_OLD_POLY1305_SHA256
- TLS_DHE_RSA_WITH_CHACHA20_OLD_POLY1305_SHA256

Renegotiation Indication Extension Special Suite:

- TLS_EMPTY_RENEGOTIATION_INFO_SCSV

4.3.3 AEAD Suites

wolfSSL supports AEAD suites, including AES-GCM, AES-CCM, and CHACHA-POLY1305. The big difference between these AEAD suites and others is that they authenticate the encrypted data with any additional cleartext data. This helps with mitigating man in the middle attacks that result in having data tampered with. AEAD suites use a combination of a block cipher (or more recently also a stream cipher) algorithm combined with a tag produced by a keyed hash algorithm. Combining these two algorithms is handled by the wolfSSL encrypt and decrypt process which makes it easier for users. All that is needed for using a specific AEAD suite is simply enabling the algorithms that are used in a supported suite.

4.3.4 Block and Stream Ciphers

wolfSSL supports the **AES**, **DES**, **3DES**, and **Camellia** block ciphers and the **RC4**, and **CHACHA20** stream ciphers. AES, DES, 3DES, RC4, and ChaCha20 are enabled by default. Camellia can be enabled when building wolfSSL with the `--enable-camellia` build option. The default mode of AES is CBC mode. To enable GCM or CCM mode with AES, use the `--enable-aesgcm` and `--enable-aesccm` build options. Please see the examples for usage and the wolfCrypt Usage Reference for specific usage information.

While SSL uses RC4 as the default stream cipher, it has been obsoleted due to compromise. Recently wolfSSL added ChaCha20. While RC4 is about 11% more performant than ChaCha, RC4 is generally considered less secure than ChaCha. ChaCha can put up very nice times of its own with added security as a tradeoff.

To see a comparison of cipher performance, visit the wolfSSL Benchmark web page, located here: https://www.wolfssl.com/docs/benchmarks.

4.3.4.1 What's the Difference? A block cipher has to be encrypted in chunks that are the block size for the cipher. For example, AES has a block size of 16 bytes. So if you're encrypting a bunch of small, 2 or 3 byte chunks back and forth, over 80% of the data is useless padding, decreasing the speed of the encryption/decryption process and needlessly wasting network bandwidth to boot. Basically, block ciphers are designed for large chunks of data, have block sizes requiring padding, and use a fixed, unvarying transformation.

Stream ciphers work well for large or small chunks of data. They are suitable for smaller data sizes because no block size is required. If speed is a concern, stream ciphers are your answer, because they use a simpler transformation that typically involves an xor'd keystream. So if you need to stream media, encrypt various data sizes including small ones, or have a need for a fast cipher then stream ciphers are your best bet.

4.3.5 Hashing Functions

wolfSSL supports several different hashing functions, including **MD2**, **MD4**, **MD5**, **SHA-1**, **SHA-2** (SHA-224, SHA-256, SHA-384, SHA-512), **SHA-3** (BLAKE2), and **RIPEMD-160**. Detailed usage of these functions can be found in the wolfCrypt Usage Reference, Hash Functions.

4.3.6 Public Key Options

wolfSSL supports the **RSA**, **ECC**, **DSA/DSS** and **DH** public key options, with support for **EDH** (Ephemeral Diffie-Hellman) on the wolfSSL server. Detailed usage of these functions can be found in the wolfCrypt Usage Reference, Public Key Cryptography.

4.3.6.1 ML-KEM, ML-DSA ML-KEM (Module Lattice Key Encapsulation Mechanism) is a NIST-standardized, lattice-based post-quantum algorithm derived from Kyber. It enables two parties to establish a shared key over an insecure channel using a key encapsulation mechanism, protecting against both classical and quantum adversaries.

ML-DSA (Module Lattice Digital Signature Algorithm) is a NIST-standardized, lattice-based post-quantum digital signature scheme derived from Dilithium. It enables a sender to produce a verifiable signature that proves the origin and integrity of a message.

Both ML-KEM and ML-DSA are public-key algorithms designed to resist cryptographically relevant quantum computers. They are part of NIST's Post-Quantum Cryptography standards (FIPS 203 and FIPS 204) and can be deployed today, often in hybrid form, to prepare for the post-quantum era.

4.3.7 ECC Support

wolfSSL has support for Elliptic Curve Cryptography (ECC) including but not limited to: ECDH-ECDSA, ECDHE-ECDSA, ECDH-RSA, ECDHE-PSK and ECDHE-RSA.

wolfSSL's ECC implementation can be found in the `<wolfssl_root>/wolfssl/wolfcrypt/ecc.h` header file and the `<wolfssl_root>/wolfcrypt/src/ecc.c` source file.

Supported cipher suites are shown in the table above. ECC is disabled by default on non x86_64 builds, but can be turned on when building wolfSSL with the HAVE_ECC define or by using the autoconf system:

```
./configure --enable-ecc
make
make check
```

When `make check` runs, note the numerous cipher suites that wolfSSL checks (if make check doesn't produce a list of cipher suites run `./testsuite/testsuite.test` on its own). Any of these cipher

suites can be tested individually, e.g., to try ECDH-ECDSA with AES256-SHA, the example wolfSSL server can be started like this:

```
./examples/server/server -d -l ECDHE-ECDSA-AES256-SHA -c
 ↪  ./certs/server-ecc.pem -k ./certs/ecc-key.pem
```

(`-d`) disables client cert check while (`-l`) specifies the cipher suite list. (`-c`) is the certificate to use and (`-k`) is the corresponding private key to use. To have the client connect try:

```
./examples/client/client -A ./certs/server-ecc.pem
```

where (`-A`) is the CA certificate to use to verify the server.

4.3.8 PKCS Support

PKCS (Public Key Cryptography Standards) refers to a group of standards created and published by RSA Security, Inc. wolfSSL has support for **PKCS #1, PKCS #3, PKCS #5, PKCS #7, PKCS #8, PKCS #9, PKCS #10, PKCS #11,** and **PKCS #12.**

Additionally, wolfSSL also provides support for RSA-Probabilistic Signature Scheme (PSS), which is standardized as part of PKCS #1.

4.3.8.1 PKCS #5, PBKDF1, PBKDF2, PKCS #12 PKCS #5 is a password-based key derivation method which combines a password, a salt, and an iteration count to generate a password-based key. wolfSSL supports both PBKDF1 and PBKDF2 key derivation functions. A key derivation function produces a derived key from a base key and other parameters (such as the salt and iteration count as explained above). PBKDF1 applies a hash function (MD5, SHA1, etc) to derive keys, where the derived key length is bounded by the length of the hash function output. With PBKDF2, a pseudorandom function is applied (such as HMAC-SHA-1) to derive the keys. In the case of PBKDF2, the derived key length is unbounded.

wolfSSL also supports the PBKDF function from PKCS #12 in addition to PBKDF1 and PBKDF2. The function prototypes look like this:

```
int wc_PBKDF2(byte* output, const byte* passwd, int pLen,
          const byte* salt, int sLen, int iterations,
          int kLen, int hashType);

int wc_PKCS12_PBKDF(byte* output, const byte* passwd, int pLen,
                const byte* salt, int sLen, int iterations,
                int kLen, int hashType, int purpose);
```

`output` contains the derived key, `passwd` holds the user password of length `pLen`, `salt` holds the salt input of length `sLen`, `iterations` is the number of iterations to perform, `kLen` is the desired derived key length, and `hashType` is the hash to use (which can be MD5, SHA1, or SHA2).

If you are using `./configure` to build wolfssl, the way to enable this functionality is to use the option `--enable-pwdbased`

A full example can be found in `<wolfSSL Root>/wolfcrypt/test.c`. More information can be found on PKCS #5, PBKDF1, and PBKDF2 from the following specifications:

PKCS#5, PBKDF1, PBKDF2: https://tools.ietf.org/html/rfc2898

4.3.8.2 PKCS #8 PKCS #8 is designed as the Private-Key Information Syntax Standard, which is used to store private key information - including a private key for some public-key algorithm and set of attributes.

The PKCS #8 standard has two versions which describe the syntax to store both encrypted private keys and non-encrypted keys. wolfSSL supports both unencrypted and encrypted PKCS #8. Supported formats include PKCS #5 version 1 - version 2, and PKCS#12. Types of encryption available include DES, 3DES, RC4, and AES.

PKCS#8: https://tools.ietf.org/html/rfc5208

4.3.8.3 PKCS #7 PKCS #7 is designed to transfer bundles of data whether as an enveloped certificate or unencrypted but signed string of data. The functionality is turned on by using the enable option (`--enable-pkcs7`) or by using the macro `HAVE_PKCS7`. Note that degenerate cases are allowed by default as per the RFC having an empty set of signers. To toggle allowing degenerate cases on and off the function `wc_PKCS7_AllowDegenerate()` can be called.

Supported features include:

- Degenerate bundles
- KARI, KEKRI, PWRI, ORI, KTRI bundles
- Detached signatures
- Compressed and Firmware package bundles
- Custom callback support
- Limited streaming capability

4.3.8.3.1 PKCS #7 Callbacks Additional callbacks and supporting functions were added to allow for a user to choose their keys after the PKCS7 bundle has been parsed. For unwrapping the CEK the function `wc_PKCS7_SetWrapCEKCb()` can be called. The callback set by this function gets called in the case of KARI and KEKRI bundles. The keyID or SKID gets passed from wolfSSL to the user along with the originator key in the case of KARI. After the user unwraps the CEK with their KEK the decrypted key to be used should then be passed back to wolfSSL. An example of this can be found in the wolfssl-examples repository in the file `signedData-EncryptionFirmwareCB.c`.

An additional callback was added for decryption of PKCS7 bundles. For setting a decryption callback function the API `wc_PKCS7_SetDecodeEncryptedCb()` can be used. To set a user defined context the API `wc_PKCS7_SetDecodeEncryptedCtx()` should be used. This callback will get executed on calls to `wc_PKCS7_DecodeEncryptedData()`.

4.3.8.3.2 PKCS #7 Streaming Stream oriented API for PKCS7 decoding gives the option of passing inputs in smaller chunks instead of all at once. By default the streaming functionality with PKCS7 is on. To turn off support for streaming PKCS7 API the macro `NO_PKCS7_STREAM` can be defined. An example of doing this with autotools would be `./configure --enable-pkcs7 CFLAGS=-DNO_PKCS7_STREAM`.

For streaming when decoding/verifying bundles the following functions are supported:

1. `wc_PKCS7_DecodeEncryptedData()`
2. `wc_PKCS7_VerifySignedData()`
3. `wc_PKCS7_VerifySignedData_ex()`
4. `wc_PKCS7_DecodeEnvelopedData()`
5. `wc_PKCS7_DecodeAuthEnvelopedData()`

Note: that when calling `wc_PKCS7_VerifySignedData_ex` it is expected that the argument pkiMsgFoot is the full buffer. The internal structure only supports streaming of one buffer, which in this case would be `pkiMsgHead`.

4.3.8.4 PKCS #12 PKCS #12 defines the PFX bundle format: a single, password-protected file that carries a private key together with its certificate and, optionally, the rest of the CA chain. It is the

format behind the .p12 and .pfx files commonly produced by OpenSSL, Java keytool, and Windows certificate export. wolfSSL implements the format as described in RFC 7292.

PKCS #12 support is enabled by using the configure option --enable-pkcs12 or by defining the macro HAVE_PKCS12. It relies on password-based key derivation, so --enable-pwdbased is required as well; both are on by default in a normal build and can be turned off with --disable-pkcs12. Older bundles are frequently encrypted with RC2 or single DES, so reading files produced by legacy tools may additionally require --enable-rc2 or --enable-des3.

The wolfCrypt API is declared in <wolfssl/wolfcrypt/pkcs12.h> and operates on an opaque WC_PKCS12 structure. All of the functions are documented in the PKCS12 API Reference.

4.3.8.4.1 Reading a PKCS #12 Bundle Reading a bundle is a three step process: allocate a WC_PKCS12 structure, decode the DER into it, then verify the MAC and decrypt the contents.

```
WC_PKCS12* pkcs12 = NULL;
byte* key  = NULL;
byte* cert = NULL;
WC_DerCertList* ca = NULL;
word32 keySz  = 0;
word32 certSz = 0;

/* wc_d2i_PKCS12_fp allocates the WC_PKCS12 when *pkcs12 is NULL */
if (wc_d2i_PKCS12_fp("./certs/test-servercert.p12", &pkcs12) != 0) {
    /* error reading or decoding the file */
}

if (wc_PKCS12_parse(pkcs12, "password", &key, &keySz, &cert, &certSz,
                    &ca) != 0) {
    /* error parsing the bundle, most commonly a wrong password */
}

/* use the DER key, cert, and ca chain here, for example with
 * wolfSSL_CTX_use_certificate_buffer() and
 * wolfSSL_CTX_use_PrivateKey_buffer() */

XFREE(key, NULL, DYNAMIC_TYPE_PUBLIC_KEY);
XFREE(cert, NULL, DYNAMIC_TYPE_PKCS);
wc_FreeCertList(ca, NULL);
wc_PKCS12_free(pkcs12);
```

If the bundle is already in memory rather than on disk, use wc_d2i_PKCS12() with a WC_PKCS12 obtained from wc_PKCS12_new() or wc_PKCS12_new_ex(). The _ex variant takes a heap hint, which is stored in the structure and used for every allocation made on its behalf.

wc_PKCS12_parse() returns the private key and the certificate in newly allocated DER buffers, and the caller owns all three results. Note that the key and the certificate are freed with different dynamic types, and that the CA list has its own free function:

Output	How to free
pkey	XFREE(pkey, heap, DYNAMIC_TYPE_PUBLIC_KEY)
cert	XFREE(cert, heap, DYNAMIC_TYPE_PKCS)
ca	wc_FreeCertList(ca, heap)

The heap above is the same heap hint that was passed to wc_PKCS12_new_ex(), or NULL when

wc_PKCS12_new() was used. The ca argument is optional; pass NULL if the extra certificates are not needed.

By default the PKCS #8 header is stripped from the returned private key. To keep it, call wc_PKCS12_parse_ex() with a non-zero keepKeyHeader argument; it is otherwise identical to wc_PKCS12_parse().

4.3.8.4.2 Creating a PKCS #12 Bundle wc_PKCS12_create() builds a bundle from a DER private key, a DER certificate, and an optional WC_DerCertList of additional certificates. The resulting structure is then serialized with wc_i2d_PKCS12().

```
WC_PKCS12* pkcs12 = NULL;
byte* der = NULL;
int derSz = 0;
char pass[] = "password";

pkcs12 = wc_PKCS12_create(pass, sizeof(pass) - 1, NULL,
                          key, keySz, cert, certSz, ca,
                          PBE_AES256_CBC,          /* key encryption  */
                          PBE_AES256_CBC,          /* cert encryption */
                          WC_PKCS12_ITT_DEFAULT,   /* encryption iterations */
                          WC_PKCS12_ITT_DEFAULT,   /* MAC iterations */
                          0, NULL);
if (pkcs12 == NULL) {
    /* error creating the bundle */
}

/* passing a NULL *der asks wc_i2d_PKCS12 to allocate the buffer */
if ((derSz = wc_i2d_PKCS12(pkcs12, &der, NULL)) <= 0) {
    /* error encoding the bundle */
}

/* write der/derSz out to a .p12 file here */

XFREE(der, NULL, DYNAMIC_TYPE_PKCS);
wc_PKCS12_free(pkcs12);
```

The nidKey and nidCert arguments choose the password-based encryption applied to the private key and to the certificate. The supported values are PBE_SHA1_RC4_128, PBE_SHA1_DES, PBE_SHA1_DES3, PBE_AES128_CBC, and PBE_AES256_CBC; passing -1 stores that part of the bundle unencrypted. PBE_AES256_CBC is recommended for new bundles, with the SHA-1 based options reserved for interoperability with older software. When iter is zero or negative, WC_PKCS12_ITT_DEFAULT (2048) iterations are used.

The name (friendlyName) and keyType arguments are accepted for API compatibility but are not currently used.

wc_i2d_PKCS12() has three modes, selected by its der and derSz arguments:

- Passing NULL for der only computes the size. The required length is stored in *derSz and LENGTH_ONLY_E is returned.
- Passing a der whose *der is NULL allocates a buffer of the right size and stores its address in *der. The caller frees it with XFREE(*der, NULL, DYNAMIC_TYPE_PKCS).
- Passing a der whose *der points to a caller-supplied buffer writes into that buffer, returning BUFFER_E if derSz indicates it is too small. Following the usual i2d convention, *der is advanced to one byte past the encoded DER on success, so keep a separate copy of the original pointer.

COPYRIGHT ©2024 wolfSSL Inc.

4.3.8.4.3 OpenSSL Compatibility Layer When the compatibility layer is enabled, the familiar OpenSSL spellings are also available from <wolfssl/openssl/pkcs12.h>: d2i_PKCS12_bio(), PKCS12_parse(), PKCS12_verify_mac(), and PKCS12_create(). These map onto wolfSSL_d2i_PKCS12_bio(), wolfSSL_PKCS12_parse(), and friends, which wrap the wolfCrypt functions described above and return WOLFSSL_X509 and WOLFSSL_EVP_PKEY objects instead of DER buffers.

4.3.9 Forcing the Use of a Specific Cipher

By default, wolfSSL will pick the "best" (highest security) cipher suite that both sides of the connection can support. To force a specific cipher, such as 128 bit AES, add something similar to:

```
wolfSSL_CTX_set_cipher_list(ctx, "AES128-SHA");
```

after the call to wolfSSL_CTX_new() so that you have:

```
ctx = wolfSSL_CTX_new(method);
wolfSSL_CTX_set_cipher_list(ctx, "AES128-SHA");
```

4.3.10 OpenQuantumSafe's liboqs Integration

Please see the appendix Experimenting with Post-Quantum Cryptography in this document for more details.

4.4 Hardware Accelerated Crypto

wolfSSL is able to take advantage of several hardware accelerated (or "assisted") crypto functionalities in various processors and chips. The following sections explain which technologies wolfSSL supports out-of-the-box.

4.4.1 AES-NI

AES is a key encryption standard used by governments worldwide, which wolfSSL has always supported. Intel has released a new set of instructions that is a faster way to implement AES. wolfSSL is the first SSL library to fully support the new instruction set for production environments.

Essentially, Intel and AMD have added AES instructions at the chip level that perform the computationally-intensive parts of the AES algorithm, boosting performance. For a list of Intel's chips that currently have support for AES-NI, you can look here:

https://ark.intel.com/search/advanced/?s=t&AESTech=true

We have added the functionality to wolfSSL to allow it to call the instructions directly from the chip, instead of running the algorithm in software. This means that when you're running wolfSSL on a chipset that supports AES-NI, you can run your AES crypto 5-10 times faster!

If you are running on an AES-NI supported chipset, enable AES-NI with the --enable-aesni build option. To build wolfSSL with AES-NI, GCC 4.4.3 or later is required to make use of the assembly code. wolfSSL supports the ASM instructions on AMD processors using the same build options.

References and further reading on AES-NI, ordered from general to specific, are listed below. For information about performance gains with AES-NI, please see the third link to the Intel Software Network page.

- AES (Wikipedia)
- AES-NI (Wikipedia)
- AES-NI (Intel Software Network page)

AES-NI will accelerate the following AES cipher modes: AES-CBC, AES-GCM, AES-CCM-8, AES-CCM, and AES-CTR. AES-GCM is further accelerated with the use of the 128-bit multiply function added to the Intel chips for the GHASH authentication.

4.4.2 STM32F2

wolfSSL is able to use the STM32F2 hardware-based cryptography and random number generator through the STM32F2 Standard Peripheral Library.

For necessary defines, see the `WOLFSSL_STM32F2` define in `settings.h`. The `WOLFSSL_STM32F2` define enables STM32F2 hardware crypto and RNG support by default. The defines for enabling these individually are `STM32F2_CRYPTO` (for hardware crypto support) and `STM32F2_RNG` (for hardware RNG support).

Documentation for the STM32F2 Standard Peripheral Library can be found in the following document: https://www.st.com/internet/com/TECHNICAL_RESOURCES/TECHNICAL_LITERATURE/USER_MANUAL/DM00023896.pdf

4.4.3 Cavium NITROX

wolfSSL has support for Marvell (previously Cavium) NITROX (https://www.marvell.com/products/security-solutions.html). To enable Marvell NITROX support when building wolfSSL use the following configure option:

```
./configure --with-cavium=/home/user/cavium/software
```

Where the `--with-cavium=**` option is pointing to your licensed cavium/software directory. Since Cavium doesn't build a library wolfSSL pulls in the `cavium_common.o` file which gives a libtool warning about the portability of this. Also, if you're using the github source tree you'll need to remove the `-Wredundant-decls` warning from the generated Makefile because the cavium headers don't conform to this warning.

Currently wolfSSL supports Cavium RNG, AES, 3DES, RC4, HMAC, and RSA directly at the crypto layer. Support at the SSL level is partial and currently just does AES, 3DES, and RC4. RSA and HMAC are slower until the Cavium calls can be utilized in non-blocking mode. The example client turns on cavium support as does the crypto test and benchmark. Please see the `HAVE_CAVIUM` define.

4.4.4 ESP32-WROOM-32

wolfSSL is able to use the ESP32-WROOM-32 hardware-based cryptography.

For necessary defines, see the `WOLFSSL_ESPWROOM32` define in `settings.h`. The `WOLFSSL_ESPWROOM32` define enables ESP32-WROOM-32 hardware crypto and RNG support by default. Currently wolfSSL supports RNG, AES, SHA and RSA primitive at the crypt layer. The example projects including TLS server/client, wolfCrypt test and benchmark can be found at /examples/protocols directory in ESP-IDF after deploying files.

4.4.5 ESP8266

Unlike the ESP32, there's no hardware-based cryptography available for the ESP8266. See the `WOLFSSL_ESP8266` define in `user_settings.h` or use `./configure CFLAGS="-DWOLFSSL_ESP8266"` to compile for the embedded ESP8266 target.

4.4.6 EFR32

wolfSSL is able to use the EFR32 family of devices for hardware-based cryptography.

To enable support define WOLFSSL_SILABS_SE_ACCEL in user_settings.h. wolfSSL currently supports the hardware acceleration of RNG, AES-CBC, AES-GCM, AES-CCM, SHA-1, SHA-2, ECDHE, and ECDSA on the EFR32 platform.

More details and benchmarks are available in the README.md in wolfcrypt/src/port/silabs of the wolfSSL repository tree.

4.4.7 MAX32665/MAX32666

wolfSSL supports using the Trust Protection Unit (TPU), Modular Arithmetic Accelerator (MAA) and TRNG found on supported models of the MAX32666/MAX32665 microcontrollers from Analog Devices.

To enable support define WOLFSSL_MAX3266X and WOLFSSL_SP_MATH_ALL. wolfSSL currently supports the hardware acceleration of RNG, AES-CBC, AES-GCM, AES-ECB, SHA-1, SHA-2, RSA 2048, and ECDSA.

This HW also supports the use of wolfSSL's crypto callback feature to allow the usage of both HW and SW implementations.

More details of the support can be found in the README.md at wolfcrypt/src/port/maxim of the wolfSSL repository tree.

4.5 SSL Inspection (Sniffer)

Beginning with the wolfSSL 1.5.0 release, wolfSSL has included a build option allowing it to be built with SSL Sniffer (SSL Inspection) functionality. This means that you can collect SSL traffic packets and with the correct key file, are able to decrypt them as well. The ability to "inspect" SSL traffic can be useful for several reasons, some of which include:

- Analyzing Network Problems
- Detecting network misuse by internal and external users
- Monitoring network usage and data in motion
- Debugging client/server communications

To enable sniffer support, build wolfSSL with the `--enable-sniffer` option on *nix or use the **vcproj** files on Windows. You will need to have **pcap** installed on *nix or **WinPcap** on Windows. The main sniffer functions which can be found in `sniffer.h` are listed below with a short description of each:

- `ssl_SetPrivateKey` - Sets the private key for a specific server and port.
- `ssl_SetNamedPrivateKey` - Sets the private key for a specific server, port and domain name.
- `ssl_DecodePacket` - Passes in a TCP/IP packet for decoding.
- `ssl_Trace` - Enables / Disables debug tracing to the traceFile.
- `ssl_InitSniffer` - Initialize the overall sniffer.
- `ssl_FreeSniffer` - Free the overall sniffer.
- `ssl_EnableRecovery` - Enables option to attempt to pick up decoding of SSL traffic in the case of lost packets.
- `ssl_GetSessionStats` - Obtains memory usage for the sniffer sessions.

To look at wolfSSL's sniffer support and see a complete example, please see the `snifftest` app in the `sslSniffer/sslSnifferTest` folder from the wolfSSL download.

Keep in mind that because the encryption keys are setup in the SSL Handshake, the handshake needs to be decoded by the sniffer in order for future application data to be decoded. For example, if you are using "snifftest" with the wolfSSL example echoserver and echoclient, the snifftest application must be started before the handshake begins between the server and client.

The sniffer can only decode streams encrypted with the following algorithms: AES-CBC, DES3-CBC, ARC4 and Camellia-CBC. If ECDHE or DHE key agreement is used the stream cannot be sniffed; only RSA or ECDH key-exchange is supported.

Watch callbacks with wolfSSL sniffer can be turned on with WOLFSSL_SNIFFER_WATCH. With the sniffer watch feature compiled in, the function ssl_SetWatchKeyCallback() can be used to set a custom callback. The callback is then used to inspect the certificate chain, error value, and digest of the certificate sent from the peer. If a non 0 value is returned from the callback then an error state is set when processing the peer's certificate. Additional supporting functions for the watch callbacks are:

- ssl_SetWatchKeyCtx: Sets a custom user context that gets passed to the watch callback.
- ssl_SetWatchKey_buffer: Loads a new DER format key into server session.
- ssl_SetWatchKey_file: File version of ssl_SetWatchKey_buffer.

Statistics collecting with the sniffer can be compiled in by defining the macro WOLFSSL_SNIFFER_STATS. The statistics are kept in a SSLStats structure and are copied to an application's SSLStats structure by a call to ssl_ReadStatistics. Additional API to use with sniffer statistics is ssl_ResetStatistics (resets the collection of statistics) and ssl_ReadResetStatistics (reads the current statistic values and then resets the internal state). The following is the current statistics kept when turned on:

- sslStandardConns
- sslClientAuthConns
- sslResumedConns
- sslEphemeralMisses
- sslResumeMisses
- sslCiphersUnsupported
- sslKeysUnmatched
- sslKeyFails
- sslDecodeFails
- sslAlerts
- sslDecryptedBytes
- sslEncryptedBytes
- sslEncryptedPackets
- sslDecryptedPackets
- sslKeyMatches
- sslEncryptedConns

4.6 Static Buffer Allocation Option

By default, wolfSSL assumes that the execution environment provides dynamic memory allocation, i.e., buffers can be allocated/freed with the malloc/free functions. The wolfCrypt cryptography library, which wolfSSL uses internally for underlying cryptography operations, can alternatively be configured to not use dynamic memory. This can be helpful for environments without dynamic memory support, or safety-critical applications where dynamic memory use is disallowed.

4.6.1 Basic Operation of Static Buffer Allocation

"Dynamic memory allocation" is a management method that dynamically finds/allocates and provides a buffer of a "specified size (variable length)". Buffer usage efficiency is high, but processing is relatively complicated. On the other hand, the "static buffer allocation" provided by wolfSSL is a memory management model that searches for a buffer close to the requested size from among several types of buffers prepared in advance (statically) and provides it back to the caller. A memory block larger than the requested size may be allocated and returned to the requester of the buffer (thus reducing the efficiency of use). Although not as precise in memory management, it is simple and simulates dynamic memory allocation that dynamically allocates a memory block of any size.

Using static-buffer-allocation is equivalent in API to using dynamic memory with wolfSSL. This functional equivalency is achieved in wolfSSL by abstracting memory allocation/free into XMALLOC/XFREE function calls. Once static-buffer-allocation is set, wolfSSL will use it from then on to allocate buffers

and other structures used internally. Since this feature is set for WOLFSSL_CTX, it will continue to work for the lifetime of the context object.

The static-buffer-allocation set in a WOLFSSL_CTX is thread-safe. Even if the same WOLFSSL_CTX is shared by different threads, buffer allocation/free is used under exclusive control inside wolfSSL. The following is a visual representation of the CTX structure, the arrows indicate passing a pointer to the heap and "..." references all structs and not to the ones listed.

In comparison to a memory pool functionality offered by an RTOS implementation, memory functionality in an RTOS will commonly suspend a thread (task) if an unused memory block cannot be found when requested until a free block becomes available. wolfSSL's static memory functionality has no such synchronization capability.

4.6.2 Specifying Static Buffer Use

With static-buffer-allocation in wolfSSL, it is possible to divide memory between two purposes. It's possible to allocate and free buffers separately between general purposes and I/O use cases. The buffer used for I/O is set relatively large (about 17KB) to accommodate the maximum TLS fragment size of up to 2^16 bytes. This is different from the buffer sizes for other general uses. Additionally, when setting the buffer configuration the user can limit the maximum number of WOLFSSL objects that can be created simultaneously. If the maximum number of WOLFSSL sessions is limited, each use of the wolfSSL_new() function will check the number of WOLFSSL objects that can be created and will error out if the limit is exceeded.

4.6.3 Enabling Static Buffer Allocation

Enable the static-buffer-allocation option when building wolfSSL. For systems built using Autoconf, specify "–enable-staticmemory" as follows:

```
$ ./configure --enable-staticmemory
```

Or if you are using a user_settings.h header, add the following macro definition:

user_settings.h

```
    #define WOLFSSL_STATIC_MEMORY
```

The static-buffer-allocation option is implemented by default to fall back to the standard malloc() function when a NULL heap hint is passed in. If a heap hint is passed in and the memory associated with it is exhausted, an error will occur. If the environment does not provide dynamic memory management functionality, a link error will occur. Therefore, also define the **WOLFSSL_NO_MALLOC** macro to disable this feature if needed:

user_settings.h

```
    #define WOLFSSL_STATIC_MEMORY
    #define WOLFSSL_NO_MALLOC
```

In addition there are two build configurations. One is `--enable-staticmemory=small` which is a smaller version that has smaller struct sizes and less supporting API's available. The other build configuration is `--enable-staticmemory`=debug that enables the ability to set a callback function. This is useful in cases where printf() is not available for determining what is being allocated and what bucket sizes are being used. Here is what the example client output looks like with example callback:

```
./examples/client/client
...
...
```

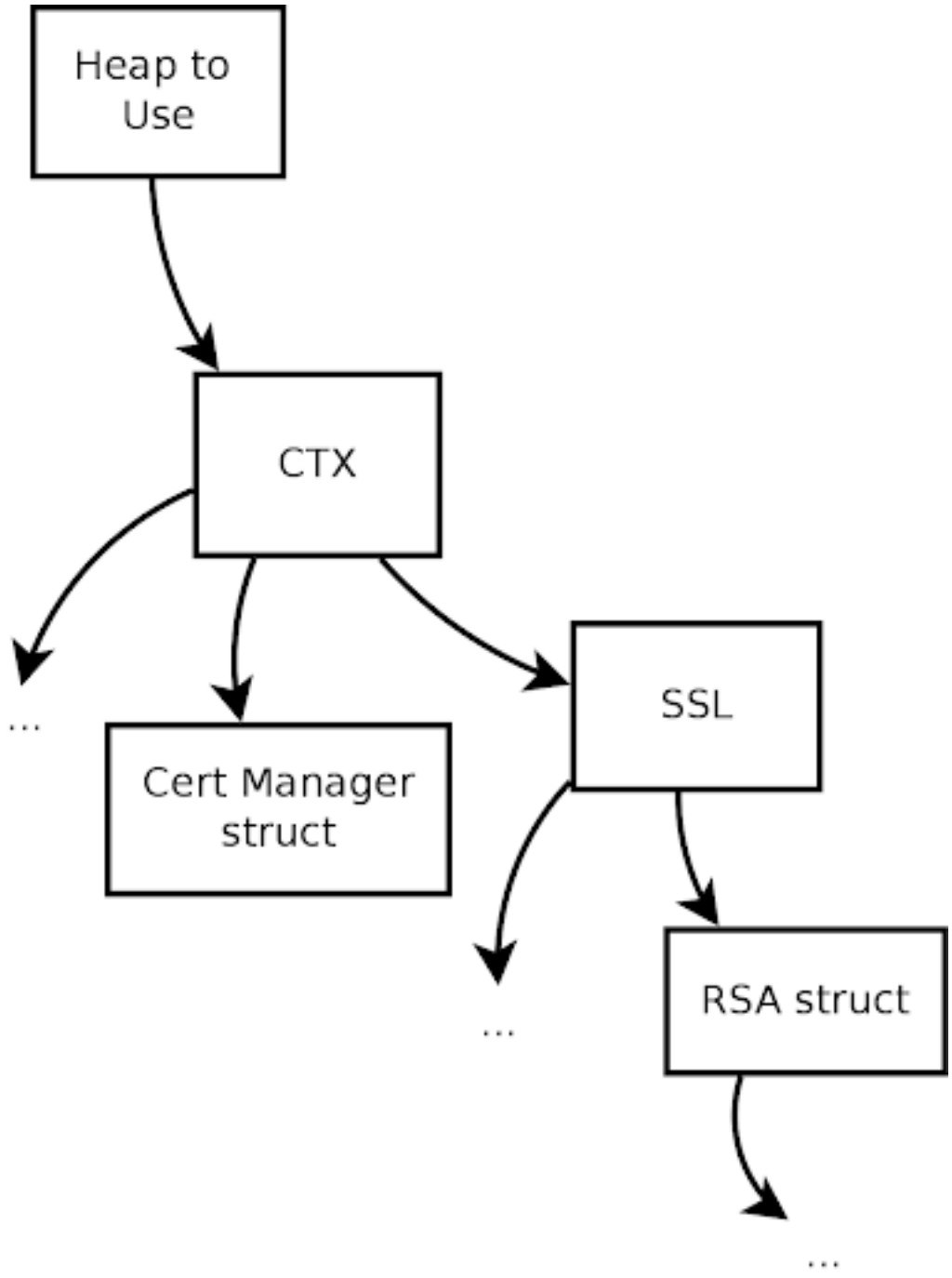

Figure 2: Alt text

```
...
Free'ing : 16128
OUT BUFFER: Alloc'd 6 bytes using bucket size 16992
Alloc'd 848 bytes using bucket size 1024
OUT BUFFER: Alloc'd 150 bytes using bucket size 16992
Alloc'd 13 bytes using bucket size 64
Alloc'd 12 bytes using bucket size 64
Alloc'd 848 bytes using bucket size 1024
IN BUFFER: Alloc'd 40 bytes using bucket size 16992
Alloc'd 13 bytes using bucket size 64
Alloc'd 12 bytes using bucket size 64
Free'ing : 1024
Free'ing : 512

...
...
...
```

4.6.4 Using Static Buffer Allocation

This can be helpful for environments without dynamic memory support, or safety-critical applications where dynamic memory use is disallowed.

4.6.4.1 Static buffer setup function and its arguments This can be helpful for environments without dynamic memory support, or safety-critical applications where dynamic memory use is disallowed.

```
int wolfSSL_CTX_load_static_memory(
      WOLFSSL_CTX** ctx, /* address of the variable to hold WOLFSSL_CTX */
      wolfSSL_method_func method,/* method pointer */
      unsigned char* buf,   /* pointer to the buffer to use as heap */
      unsigned int sz,      /* buffer size */
      int flag,              /* heap usage */
      int max);              /* maximum number of objects allowed */
```

- parameter **ctx** specifies the address of a variable that receives a pointer to the generated WOLFSSL_CTX structure.
- parameter **method** specifies a function pointer with "_ex", such as wolfSSLv23_client_method_ex(). The functions that can be used are listed in a later chapter.
- parameter **buf** and **sz** specify the address and size of the buffer used for the heap, respectively. For information on determining the buffer size to be set, see "**Obtaining the Required Buffer Size**".
- parameter **flag** is a flag that specifies the usage of the buffer. You can also specify whether to track the allocation status. When specifying for general use, specify "**0**" or **WOLFMEM_GENERAL**. For I/O use, specify **WOLFMEM_IO_POOL** or **WOLFMEM_IO_POOL_FIXED**. When tracking the allocation status of static buffers, **OR** the value specifying the usage with **WOLFMEM_TRACK_STATS**.
- parameter **max** is related to the use of the buffer specified by the argument flag. If the buffer is for general use, you may want to set the maximum number of WOLFSSL objects that can be generated simultaneously (the number of objects that can exist at the same time). Specify **0** if there is **no need to limit**. If you specify a limit value other than 0, subsequent calls to wolfSSL_new() will fail if the number of concurrent WOLFSSL objects created exceeds the set value.

4.6.4.2 How to call the static buffer setup function When using the static-buffer-allocation option, call the wolfSSL_CTX_load_static_memory() function **twice**. The first sets up a buffer for general use, and then uses that buffer to allocate a WOLFSSL_CTX structure. The second call sets up the I/O buffer:

```
WOLFSSL_CTX* ctx = NULL; /* pass NULL to generate WOLFSSL_CTX */
int ret;

#define MAX_CONCURRENT_TLS  0
#define MAX_CONCURRENT_IO   0

unsigned char GEN_MEM[GEN_MEM_SIZE];
unsigned char IO_MEM[IO_MEM_SIZE];

  /* set up a general-purpose buffer and generate WOLFSSL_CTX from it on the
     first call. */
   ret = wolfSSL_CTX_load_static_memory(
           &ctx,                                 /* set NULL to ctx */
           wolfSSLv23_client_method_ex(),  /* use function with "_ex" */
           GEN_MEM, GEN_MEM_SIZE,            /* buffer and its size */
           WOLFMEM_GENERAL,                  /* general purpose */
           MAX_CONCURRENT_TLS);              /* max concurrent objects */

   /* set up a I/O-purpose buffer on the second call. */
   ret = wolfSSL_CTX_load_static_memory(
           &ctx,                /* make sure ctx is holding the object */
           NULL,                             /* pass it to NULL this time */
           IO_MEM, IO_MEM_SIZE,                /* buffer and its size */
           WOLFMEM_IO_FIXED,                                 /* I/O purpose */
           MAX_CONCURRENT_IO);               /* max concurrent objects */
```

After this, when you are done using the WOLFSSL_CTX structure, free it with the usual wolfSSL_CTX_free().

4.6.5 Using Static Memory Without TLS (Crypto-Only)

wolfSSL also supports using static memory allocation for wolfCrypt operations without requiring TLS functionality. This is useful for applications that only need cryptographic operations (hashing, encryption, random number generation, etc.) without the overhead of TLS connections.

The `wc_LoadStaticMemory()` function is used to set aside static memory for wolfCrypt use. Memory can be used by passing the created heap hint into functions that support it. An example of this is when calling `wc_InitRng_ex()`.

Example:

```
WOLFSSL_HEAP_HINT hint;
int ret;
unsigned char memory[MAX];
int memorySz = MAX;
int flag = WOLFMEM_GENERAL;

// load in memory for use
ret = wc_LoadStaticMemory(&hint, memory, memorySz, flag, 0);
if (ret != SSL_SUCCESS) {
    // handle error case
```

```
}

ret = wc_InitRng_ex(&rng, hint, 0);
// check ret value
```

Example with Global Heap Hint:

```
WOLFSSL_HEAP_HINT hint;
void* oldHint;
int ret;
unsigned char memory[MAX];

// Set up static memory
ret = wc_LoadStaticMemory(&hint, memory, MAX, WOLFMEM_GENERAL, 0);
if (ret != SSL_SUCCESS) {
    // handle error case
}

// Set as global heap hint (not thread safe)
oldHint = wolfSSL_SetGlobalHeapHint(hint);

// Now all NULL heap hint calls will use this global hint
// ... use wolfSSL functions ...

// Restore previous global heap hint when done
wolfSSL_SetGlobalHeapHint(oldHint);
wc_UnloadStaticMemory(hint);
```

Example with Custom Buffer Sizes:

```
WOLFSSL_HEAP_HINT hint;
word32 customSizes[] = {64, 128, 256, 512, 1024};
word32 customDist[] = {10, 5, 3, 2, 1};
int requiredSize, ret;
unsigned char memory[MAX];

// Calculate required buffer size for custom configuration
requiredSize = wolfSSL_StaticBufferSz_ex(5, customSizes, customDist,
                                         NULL, 0, WOLFMEM_GENERAL);
printf("Required buffer size: %d bytes\n", requiredSize);

// Allocate memory buffer
if (requiredSize <= MAX) {
    // Load static memory with custom bucket configuration
    ret = wc_LoadStaticMemory_ex(&hint, 5, customSizes, customDist,
                                 memory, requiredSize, WOLFMEM_GENERAL, 0);
    if (ret != SSL_SUCCESS) {
        // handle error case
    }

    // Use the custom-configured static memory
    // ... use wolfSSL functions with hint ...

    // Clean up when done
    wc_UnloadStaticMemory(hint);
}
```

```
// Calculate padding overhead
int paddingSize = wolfSSL_MemoryPaddingSz();
printf("Memory padding per bucket: %d bytes\n", paddingSize);
```

4.6.5.1 Setting a Global Heap Hint A global heap hint can be set using the API `void* wolfSSL_SetGlobalHeapHint(void* heap)`. When a global heap hint is set all calls to XMALLOC and XFREE using a NULL pointer as the heap hint will be redirected to use the global heap hint set. This is useful in cases where no system malloc is available to fall back to and NULL heap hint pointers are being used. The function wolfSSL_SetGlobalHeapHint returns the current global heap hint set, and is NOT considered to be thread safe.

The getter function `void* wolfSSL_GetGlobalHeapHint(void)` can be used to get the current global heap hint set.

This functionality was added in versions of wolfSSL after version 5.7.0.

4.6.5.2 Debug Memory Callback When using static memory allocation with the `WOLFSSL_STATIC_MEMORY_DEBUG_CALLBACK` build option, you can set a debug callback function using `void wolfSSL_SetDebugMemoryCb(DebugMemoryCb cb)`. This callback is called during memory allocation and deallocation operations to provide debugging information about memory usage.

The callback function signature is:

```
typedef void (*DebugMemoryCb)(size_t sz, int bucketSz, byte st, int type);
```

Where: - `sz` is the requested size - `bucketSz` is the bucket size used - `st` is the status (0=alloc, 1=fail, 2=free, 3=init) - `type` is the memory type

4.6.5.3 Unloading Static Memory When you're done using static memory allocation, you should call `void wc_UnloadStaticMemory(WOLFSSL_HEAP_HINT* heap)` to properly free the static memory heap and associated mutex. This ensures proper cleanup of resources.

4.6.5.4 Advanced Static Buffer Size Calculation The function `int wolfSSL_StaticBufferSz_ex(unsigned int listSz, const word32 *sizeList, const word32 *distList, byte* buffer, word32 sz, int flag)` is a pre-allocation planning tool that calculates how much of a given buffer can be effectively used for static memory allocation, based on the provided bucket sizes and distribution. It returns the usable portion of the buffer that fits complete buckets, excluding any unusable leftover space. This is useful for sizing and tuning static memory pools to maximize utilization and reduce overall memory usage.

Parameters: - `listSz`: Number of bucket sizes in the list - `sizeList`: Array of bucket sizes - `distList`: Array of distribution counts for each bucket size - `buffer`: Buffer to test (can be NULL for calculation only) - `sz`: Size of the buffer - `flag`: Memory flag (WOLFMEM_GENERAL, WOLFMEM_IO_POOL, etc.)

4.6.5.5 Memory Padding Calculation The function `int wolfSSL_MemoryPaddingSz(void)` calculates the size of management memory needed for each memory bucket, including alignment padding. This is useful for understanding the overhead of static memory allocation and for precise memory planning.

4.6.6 Adjustment of Static Buffer Allocation

The static-buffer-allocation option provided by wolfSSL manages the specified buffer by dividing it into multiple areas called "buckets" as shown in the following diagram. Multiple memory blocks of the

same size are linked within a bucket. The figure below omits the structure that manages the memory block, but a buffer with a size that includes the omitted structure is required.

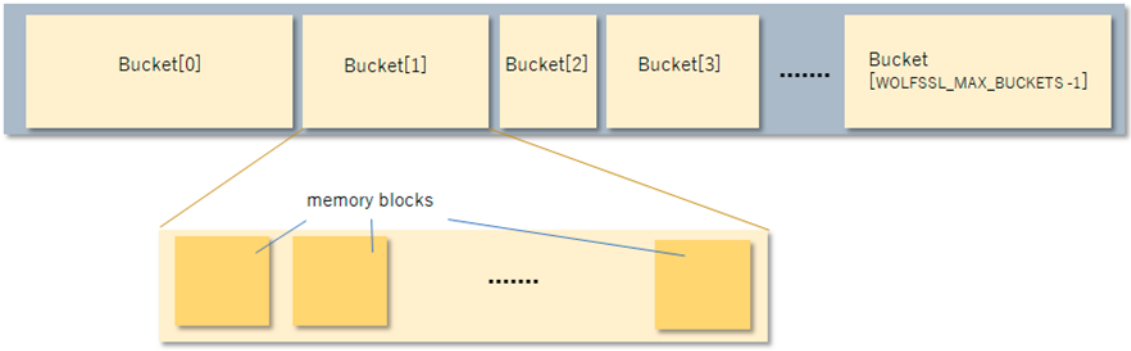

Figure 3: Alt text

4.6.6.1 Macros for General Use Buffers Each bucket varies in size depending on the number of memory blocks it contains and their size.

The memory block size and number of blocks for each area to be used are defined in /wolfssl/wolfcrypt/memory.h with the following macros:

`/wolfssl/wolfcrypt/memory.h`

```
#define WOLFSSL_STATIC_ALIGN 16  /* alignment 16 bytes by default*/
#define WOLFMEM_MAX_BUCKETS  9   /* number of buckets */
#define WOLFMEM_IO_SZ        16992   /* buffer size for I/O  */
#define LARGEST_MEM_BUCKET   16128   /* the max block size */
#define WOLFMEM_BUCKETS      64,128,256,512,1024,2432,3456,
                                                  4544,LARGEST_MEM_BUCKET
#define WOLFMEM_DIST         49,10,6,14,5,6,9,1,1
```

- **WOLFSSL_STATIC_ALIGN** specifies the buffer alignment size. It is 16 bytes by default. You need to change it according to the alignment size of your MCU.
- **WOLFMEM_MAX_BUCKETS** shows the number of buckets. This means using 9 different bucket sizes.
- **WOLFMEM_BUCKETS** specifies the number of bytes in blocks in each bucket, separated by commas, from smallest to largest. This definition applies to general purpose buffers.
- **WOLFMEM_DIST** specifies the number of same-sized blocks in each bucket, separated by commas, corresponding to each block in WOLFMEM_BUCKETS. This definition applies to general purpose buffers.

In the example above, a bucket with a block size of 64 bytes is the minimum size, and that bucket would have 49 memory blocks. The next larger bucket means 10 memory blocks with a block size of 128 bytes. The above defined values can be used as default values, but the size of each bucket and the number of memory blocks it contains may need to be adjusted when used in an actual environment.

4.6.6.2 Macros for I/O Use Buffers For TLS client:

- `wolfTLSv1_3_client_method_ex`
- `wolfTLSv1_2_client_method_ex`
- `wolfTLSv1_1_client_method_ex`

- wolfSSLv23_client_method_ex

For TLS server:

- wolfTLSv1_3_server_method_ex
- wolfTLSv1_2_server_method_ex
- wolfTLSv1_1_server_method_ex
- wolfSSLv23_server_method_ex

For DTLS client:

- wolfDTLSv1_3_client_method_ex
- wolfTLSv1_2_client_method_ex
- wolfTLSv1_1_client_method_ex
- wolfSSLv23_client_method_ex

For DTLS server:

- wolfDTLSv1_3_server_method_ex
- wolfTLSv1_2_server_method_ex
- wolfTLSv1_1_server_method_ex
- wolfSSLv23_server_method_ex

4.6.6.3 APIs for Static Buffer Allocation

API	description
wolfSSL_CTX_load_static_memory	Set buffer for WOLFSSL_CTX as a heap memory.
wolfSSL_CTX_is_static_memory	Returns whether “Static buffer Allocation” is used. If it is the case, gets usage report.
wolfSSL_is_static_memory	Returns whether “Static buffer Allocation” is used. If it is the case, gets usage report.
wc_LoadStaticMemory	Used to set aside static memory for wolfCrypt use.
wc_LoadStaticMemory_ex	Used to set aside static memory for wolfCrypt use with custom bucket sizes and distributions.
wolfSSL_SetGlobalHeapHint	Set a global heap hint that will be used when NULL heap hint is passed to memory allocation functions. Returns the previous global heap hint.
wolfSSL_GetGlobalHeapHint	Get the current global heap hint that is used when NULL heap hint is passed to memory allocation functions.
wolfSSL_SetDebugMemoryCb	Set a debug callback function for static memory allocation tracking. Used with WOLFSSL_STATIC_MEMORY_DEBUG_CALLBACK build option.
wc_UnloadStaticMemory	Free static memory heap and associated mutex. Should be called when done using static memory allocation.
wolfSSL_StaticBufferSz	Calculate required buffer size for “Static buffer Allocation” based on the macros defined in /wolfssl/wolfcrypt/memory.h.
wolfSSL_StaticBufferSz_ex	Calculate required buffer size for static memory allocation with custom bucket sizes and distributions.

API	description
wolfSSL_MemoryPaddingSz	Calculate the size of management memory needed for each memory bucket, including alignment padding.

4.7 Compression

wolfSSL supports data compression with the **zlib** library. The ./configure build system detects the presence of this library, but if you're building in some other way define the constant HAVE_LIBZ and include the path to zlib.h for your includes.

Compression is off by default for a given cipher. To turn it on, use the function wolfSSL_set_compression() before SSL connecting or accepting. Both the client and server must have compression turned on in order for compression to be used.

Keep in mind that while compressing data before sending decreases the actual size of the messages being sent and received, the amount of data saved by compression usually takes longer time to analyze than it does to send it raw on all but the slowest of networks.

4.8 Pre-Shared Keys

wolfSSL has support for these ciphers with static pre-shared keys:

- TLS_PSK_WITH_AES_256_CBC_SHA
- TLS_PSK_WITH_AES_128_CBC_SHA256
- TLS_PSK_WITH_AES_256_CBC_SHA384
- TLS_PSK_WITH_AES_128_CBC_SHA
- TLS_PSK_WITH_NULL_SHA256
- TLS_PSK_WITH_NULL_SHA384
- TLS_PSK_WITH_NULL_SHA
- TLS_PSK_WITH_AES_128_GCM_SHA256
- TLS_PSK_WITH_AES_256_GCM_SHA384
- TLS_PSK_WITH_AES_128_CCM
- TLS_PSK_WITH_AES_256_CCM
- TLS_PSK_WITH_AES_128_CCM_8
- TLS_PSK_WITH_AES_256_CCM_8
- TLS_PSK_WITH_CHACHA20_POLY1305

These suites are built into wolfSSL with WOLFSSL_STATIC_PSK on, all PSK suites can be turned off at build time with the constant NO_PSK. To only use these ciphers at runtime use the function wolfSSL_CTX_set_cipher_list() with the desired ciphersuite.

wolfSSL has support for ephemeral key PSK suites:

- ECDHE-PSK-AES128-CBC-SHA256
- ECDHE-PSK-NULL-SHA256
- ECDHE-PSK-CHACHA20-POLY1305
- DHE-PSK-CHACHA20-POLY1305
- DHE-PSK-AES256-GCM-SHA384
- DHE-PSK-AES128-GCM-SHA256
- DHE-PSK-AES256-CBC-SHA384
- DHE-PSK-AES128-CBC-SHA256
- DHE-PSK-AES128-CBC-SHA256

On the client, use the function wolfSSL_CTX_set_psk_client_callback() to setup the callback. The client example in <wolfSSL_Home>/examples/client/client.c gives example usage for setting up the client identity and key, though the actual callback is implemented in wolfssl/test.h.

On the server side two additional calls are required:

- wolfSSL_CTX_set_psk_server_callback()
- wolfSSL_CTX_use_psk_identity_hint()

The server stores its identity hint to help the client with the 2nd call, in our server example that's "wolfssl server". An example server psk callback can also be found in my_psk_server_cb() in wolfssl/test.h.

wolfSSL supports identities and hints up to 128 octets and pre-shared keys up to 64 octets.

4.9 Client Authentication

Client authentication is a feature which enables the server to authenticate clients by requesting that the clients send a certificate to the server for authentication when they connect. Client authentication requires an X.509 client certificate from a CA (or self-signed if generated by you or someone other than a CA).

By default, wolfSSL validates all certificates that it receives - this includes both client and server. To set up client authentication, the server must load the list of trusted CA certificates to be used to verify the client certificate against:

```
wolfSSL_CTX_load_verify_locations(ctx, caCert, 0);
```

To turn on client verification and control its behavior, the wolfSSL_CTX_set_verify() function is used. In the following example, SSL_VERIFY_PEER turns on a certificate request from the server to the client. SSL_VERIFY_FAIL_IF_NO_PEER_CERT instructs the server to fail if the client does not present a certificate to validate on the server side. Other options to wolfSSL_CTX_set_verify() include SSL_VERIFY_NONE and SSL_VERIFY_CLIENT_ONCE.

```
wolfSSL_CTX_set_verify(ctx, SSL_VERIFY_PEER | (usePskPlus ?
                       SSL_VERIFY_FAIL_EXCEPT_PSK :
                       SSL_VERIFY_FAIL_IF_NO_PEER_CERT), 0);
```

An example of client authentication can be found in the example server (server.c) included in the wolfSSL download (/examples/server/server.c).

4.10 Server Name Indication

SNI is useful when a server hosts multiple 'virtual' servers at a single underlying network address. It may be desirable for clients to provide the name of the server which it is contacting. To enable SNI with wolfSSL you can simply do:

```
./configure --enable-sni
```

Using SNI on the client side requires an additional function call, which should be one of the following functions:

- wolfSSL_CTX_UseSNI()
- wolfSSL_UseSNI()

wolfSSL_CTX_UseSNI() is most recommended when the client contacts the same server multiple times. Setting the SNI extension at the context level will enable the SNI usage in all SSL objects created from that same context from the moment of the call forward.

wolfSSL_UseSNI() will enable SNI usage for one SSL object only, so it is recommended to use this function when the server name changes between sessions.

On the server side one of the same function calls is required. Since the wolfSSL server doesn't host multiple 'virtual' servers, the SNI usage is useful when the termination of the connection is desired in the case of SNI mismatch. In this scenario, `wolfSSL_CTX_UseSNI()` will be more efficient, as the server will set it only once per context creating all subsequent SSL objects with SNI from that same context.

4.11 Handshake Modifications

4.11.1 Grouping Handshake Messages

wolfSSL has the ability to group handshake messages if the user desires. This can be done at the context level with `wolfSSL_CTX_set_group_messages(ctx);` or at the SSL object level with `wolfSSL_set_group_messages(ssl);`.

4.12 Truncated HMAC

Currently defined TLS cipher suites use the HMAC to authenticate record-layer communications. In TLS, the entire output of the hash function is used as the MAC tag. However, it may be desirable in constrained environments to save bandwidth by truncating the output of the hash function to 80 bits when forming MAC tags. To enable the usage of Truncated HMAC at wolfSSL you can simply do:

```
./configure --enable-truncatedhmac
```

Using Truncated HMAC on the client side requires an additional function call, which should be one of the following functions:

- `wolfSSL_CTX_UseTruncatedHMAC()`
- `wolfSSL_UseTruncatedHMAC()`

`wolfSSL_CTX_UseTruncatedHMAC()` is most recommended when the client would like to enable Truncated HMAC for all sessions. Setting the Truncated HMAC extension at context level will enable it in all SSL objects created from that same context from the moment of the call forward.

`wolfSSL_UseTruncatedHMAC()` will enable it for one SSL object only, so it's recommended to use this function when there is no need for Truncated HMAC on all sessions.

On the server side no call is required. The server will automatically attend to the client's request for Truncated HMAC.

All TLS extensions can also be enabled with:

```
./configure --enable-tlsx
```

4.13 Timing-Resistance in wolfSSL

wolfSSL provides the function "ConstantCompare" which guarantees constant time when doing comparison operations that could potentially leak timing information. This API is used at both the TLS and crypto level in wolfSSL to deter against timing based, side-channel attacks.

The wolfSSL ECC implementation has the define `ECC_TIMING_RESISTANT` to enable timing-resistance in the ECC algorithm. Similarly the define `TFM_TIMING_RESISTANT` is provided in the fast math libraries for RSA algorithm timing-resistance. The function exptmod uses the timing resistant Montgomery ladder.

See also: `--disable-harden`

Timing resistance and cache resistance defines enabled with `--enable-harden`:

- DEPRECATED: `WOLFSSL_SP_CACHE_RESISTANT`: Always on by default, see `--disable-harden` to disable default cache resistance.

- WC_RSA_BLINDING: Enables blinding mode, to prevent timing attacks.
- ECC_TIMING_RESISTANT: ECC specific timing resistance.
- TFM_TIMING_RESISTANT: Fast math specific timing resistance.

4.14 Fixed ABI

wolfSSL provides a fixed Application Binary Interface (ABI) for a subset of the Application Programming Interface (API). Starting with wolfSSL v4.3.0, the following functions will be compatible across all future releases of wolfSSL:

- wolfSSL_Init()
- wolfTLSv1_2_client_method()
- wolfTLSv1_3_client_method()
- wolfSSL_CTX_new()
- wolfSSL_CTX_load_verify_locations()
- wolfSSL_new()
- wolfSSL_set_fd()
- wolfSSL_connect()
- wolfSSL_read()
- wolfSSL_write()
- wolfSSL_get_error()
- wolfSSL_shutdown()
- wolfSSL_free()
- wolfSSL_CTX_free()
- wolfSSL_check_domain_name()
- wolfSSL_UseALPN()
- wolfSSL_CTX_SetMinVersion()
- wolfSSL_pending()
- wolfSSL_set_timeout()
- wolfSSL_CTX_set_timeout()
- wolfSSL_get_session()
- wolfSSL_set_session()
- wolfSSL_flush_sessions()
- wolfSSL_CTX_set_session_cache_mode()
- wolfSSL_get_sessionID()
- wolfSSL_UseSNI()
- wolfSSL_CTX_UseSNI()
- wc_ecc_init_ex()
- wc_ecc_make_key_ex()
- wc_ecc_sign_hash()
- wc_ecc_free()
- wolfSSL_SetDevId()
- wolfSSL_CTX_SetDevId()
- wolfSSL_CTX_SetEccSignCb()
- wolfSSL_CTX_use_certificate_chain_file()
- wolfSSL_CTX_use_certificate_file()
- wolfSSL_use_certificate_chain_file()
- wolfSSL_use_certificate_file()
- wolfSSL_CTX_use_PrivateKey_file()
- wolfSSL_use_PrivateKey_file()
- wolfSSL_X509_load_certificate_file()
- wolfSSL_get_peer_certificate()
- wolfSSL_X509_NAME_oneline()
- wolfSSL_X509_get_issuer_name()

- wolfSSL_X509_get_subject_name()
- wolfSSL_X509_get_next_altname()
- wolfSSL_X509_notBefore()
- wolfSSL_X509_notAfter()
- wc_ecc_key_new()
- wc_ecc_key_free()

COPYRIGHT ©2024 wolfSSL Inc.

5 Portability

5.1 Abstraction Layers

5.1.1 C Standard Library Abstraction Layer

wolfSSL (formerly CyaSSL) can be built without the C standard library to provide a higher level of portability and flexibility to developers. The user will have to map the functions they wish to use instead of the C standard ones.

5.1.1.1 Memory Use Most C programs use `malloc()` and `free()` for dynamic memory allocation. wolfSSL uses `XMALLOC()` and `XFREE()` instead. By default, these point to the C runtime versions. By defining `XMALLOC_USER`, the user can provide their own hooks. Each memory function takes two additional arguments over the standard ones, a heap hint, and an allocation type. The user is free to ignore these or use them in any way they like. You can find the wolfSSL memory functions in `wolfssl/wolfcrypt/types.h`.

wolfSSL also provides the ability to register memory override functions at runtime instead of compile time. `wolfssl/wolfcrypt/memory.h` is the header for this functionality and the user can call the following function to set up the memory functions:

```
int wolfSSL_SetAllocators(wolfSSL_Malloc_cb  malloc_function,
                          wolfSSL_Free_cb    free_function,
                          wolfSSL_Realloc_cb realloc_function);
```

See the header `wolfssl/wolfcrypt/memory.h` for the callback prototypes and `memory.c` for the implementation.

5.1.1.2 string.h wolfSSL uses several functions that behave like `string.h`'s `memcpy()`, `memset()`, and `memcmp()` amongst others. They are abstracted to `XMEMCPY()`, `XMEMSET()`, and `XMEMCMP()` respectively. And by default, they point to the C standard library versions. Defining `STRING_USER` allows the user to provide their own hooks in `types.h`. For example, by default `XMEMCPY()` is:

```
#define XMEMCPY(d,s,l)    memcpy((d),(s),(l))
```

After defining `STRING_USER` you could do:

```
#define XMEMCPY(d,s,l)    my_memcpy((d),(s),(l))
```

Or if you prefer to avoid macros:

```
extern void* my_memcpy(void* d, const void* s, size_t n);
```

to set wolfSSL's abstraction layer to point to your version `my_memcpy()`.

5.1.1.3 math.h wolfSSL uses two functions that behave like `math.h`'s `pow()` `log()`. They are only required by Diffie-Hellman, so if you exclude DH from the build, then you don't have to provide your own. They are abstracted to `XPOW()` and `XLOG()` and found in `wolfcrypt/src/dh.c`.

5.1.1.4 File System Use By default, wolfSSL uses the system's file system for the purpose of loading keys and certificates. This can be turned off by defining `NO_FILESYSTEM`, see item V. If instead, you'd like to use a file system but not the system one, you can use the `XFILE()` layer in `ssl.c` to point the file system calls to the ones you'd like to use. See the example provided by the MICRIUM define.

5.1.2 Custom Input/Output Abstraction Layer

wolfSSL provides a custom I/O abstraction layer for those who wish to have higher control over I/O of their SSL connection or run SSL on top of a different transport medium other than TCP/IP.

The user will need to define two functions:

1. The network Send function
2. The network Receive function

These two functions are prototyped by `CallbackIOSend` and `CallbackIORecv` in `wolfssl/wolfio.h`:

```
typedef int (*CallbackIORecv)(WOLFSSL *ssl, char *buf, int sz, void *ctx);
typedef int (*CallbackIOSend)(WOLFSSL *ssl, char *buf, int sz, void *ctx);
```

The user needs to register these functions per `WOLFSSL_CTX` with `wolfSSL_SetIOSend()` and `wolfSSL_SetIORecv()`. For example, in the default case, `CBIORecv()` and `CBIOSend()` are registered at the bottom of `io.c`:

```
void wolfSSL_SetIORecv(WOLFSSL_CTX *ctx, CallbackIORecv CBIORecv)
{
     ctx->CBIORecv = CBIORecv;
}

void wolfSSL_SetIOSend(WOLFSSL_CTX *ctx, CallbackIOSend CBIOSend)
{
    ctx->CBIOSend = CBIOSend;
}
```

The user can set a context per WOLFSSL object (session) with `wolfSSL_SetIOWriteCtx()` and `wolfSSL_SetIOReadCtx()`, as demonstrated at the bottom of `io.c`. For example, if the user is using memory buffers, the context may be a pointer to a structure describing where and how to access the memory buffers. The default case, with no user overrides, registers the socket as the context.

The `CBIORecv` and `CBIOSend` function pointers can be pointed to your custom I/O functions. The default `Send()` and `Receive()` functions, `EmbedSend()` and `EmbedReceive()`, located in `io.c`, can be used as templates and guides.

`WOLFSSL_USER_IO` can be defined to remove the automatic setting of the default I/O functions `EmbedSend()` and `EmbedReceive()`.

5.1.3 Operating System Abstraction Layer

The wolfSSL OS abstraction layer helps facilitate easier porting of wolfSSL to a user's operating system. The `wolfssl/wolfcrypt/settings.h` file contains settings which end up triggering the OS layer.

OS-specific defines are located in `wolfssl/wolfcrypt/types.h` for wolfCrypt and `wolfssl/internal.h` for wolfSSL.

5.2 Supported Operating Systems

One factor which defines wolfSSL is its ability to be easily ported to new platforms. As such, wolfSSL has support for a long list of operating systems out-of-the-box. Currently-supported operating systems include:

- Win32/64
- Linux
- Mac OS X
- Solaris

- ThreadX
- VxWorks
- FreeBSD
- NetBSD
- OpenBSD
- embedded Linux
- Yocto Linux
- OpenEmbedded
- WinCE
- Haiku
- OpenWRT
- iPhone (iOS)
- Android
- Nintendo Wii and Gamecube through DevKitPro
- QNX
- MontaVista
- NonStop
- TRON/ITRON/µITRON
- Micrium's µC/OS-III
- FreeRTOS
- SafeRTOS
- NXP/Freescale MQX
- Nucleus
- TinyOS
- HP/UX
- AIX
- ARC MQX
- TI-RTOS
- uTasker
- embOS
- INtime
- Mbed
- µT-Kernel
- RIOT
- CMSIS-RTOS
- FROSTED
- Green Hills INTEGRITY
- Keil RTX
- TOPPERS
- PetaLinux
- Apache Mynewt

5.3 Supported Chipmakers

wolfSSL has support for chipsets including ARM, Intel, Motorola, mbed, Freescale, Microchip (PIC32), STMicro (STM32F2/F4), NXP, Analog Devices, Texas Instruments, AMD and more.

5.4 C# Wrapper

wolfSSL has limited support for use in C#. A Visual Studio project containing the port can be found in the directory `root_wolfSSL/wrapper/CSharp/`. After opening the Visual Studio project set the "Active solution configuration" and "Active solution platform" by clicking on BUILD->Configuration Manager... The supported "Active solution configuration"s are DLL Debug and DLL Release. The supported platforms are Win32 and x64.

Once having set the solution and platform the preprocessor flag HAVE_CSHARP will need to be added. This turns on the options used by the C# wrapper and used by the examples included.

To then build simply select build solution. This creates the `wolfssl.dll`, `wolfSSL_CSharp.dll` and examples. Examples can be ran by targeting them as an entry point and then running debug in Visual Studio.

Adding the created C# wrapper to C# projects can be done a couple of ways. One way is to install the created `wolfssl.dll` and `wolfSSL_CSharp.dll` into the directory `C:/Windows/System/`. This will allow projects that have:

```
using wolfSSL.CSharp

public some_class {

    public static main(){
    wolfssl.Init()
    ...
}
...
```

to make calls to the wolfSSL C# wrapper. Another way is to create a Visual Studio project and have it reference the bundled C# wrapper solution in wolfSSL.

6 Callbacks

6.1 HandShake Callback

wolfSSL (formerly CyaSSL) has an extension that allows a HandShake Callback to be set for connect or accept. This can be useful in embedded systems for debugging support when another debugger isn't available and sniffing is impractical. To use wolfSSL HandShake Callbacks, use the extended functions, `wolfSSL_connect_ex()` and `wolfSSL_accept_ex()`:

```
int wolfSSL_connect_ex(WOLFSSL*, HandShakeCallBack, TimeoutCallBack,
                       Timeval)
int wolfSSL_accept_ex(WOLFSSL*, HandShakeCallBack, TimeoutCallBack,
                      Timeval)
```

HandShakeCallBack is defined as:

```
typedef int (*HandShakeCallBack)(HandShakeInfo*);
```

HandShakeInfo is defined in `wolfssl/callbacks.h` (which should be added to a non-standard build):

```
typedef struct handShakeInfo_st {
    char   cipherName[MAX_CIPHERNAME_SZ + 1];  /*negotiated name */
    char   packetNames[MAX_PACKETS_HANDSHAKE][MAX_PACKETNAME_SZ+1];
                                          /* SSL packet names */
    int    numberPackets;                 /*actual # of packets  */
    int    negotiationError;              /*cipher/parameter err */
} HandShakeInfo;
```

No dynamic memory is used since the maximum number of SSL packets in a handshake exchange is known. Packet names can be accessed through `packetNames[idx]` up to `numberPackets`. The callback will be called whether or not a handshake error occurred. Example usage is also in the client example.

6.2 Timeout Callback

The same extensions used with wolfSSL Handshake Callbacks can be used for wolfSSL Timeout Callbacks as well. These extensions can be called with either, both, or neither callbacks (Handshake and/or Timeout). `TimeoutCallback` is defined as:

```
typedef int (*TimeoutCallBack)(TimeoutInfo*);
```

Where `TimeoutInfo` looks like:

```
typedef struct timeoutInfo_st {
    char       timeoutName[MAX_TIMEOUT_NAME_SZ +1]; /*timeout Name*/
    int        flags;                               /* for future use*/
    int        numberPackets;                /*actual # of packets */
    PacketInfo packets[MAX_PACKETS_HANDSHAKE];  /*list of packets */
    Timeval    timeoutValue;                /*timer that caused it */
} TimeoutInfo;
```

Again, no dynamic memory is used for this structure since a maximum number of SSL packets is known for a handshake. `Timeval` is just a typedef for struct timeval.

`PacketInfo` is defined like this:

```
typedef struct packetInfo_st {
    char           packetName[MAX_PACKETNAME_SZ + 1]; /* SSL name */
    Timeval        timestamp;                  /* when it occurred */
```

```
    unsigned char  value[MAX_VALUE_SZ];     /* if fits, it's here */
    unsigned char* bufferValue;         /* otherwise here (non 0) */
    int            valueSz;              /* sz of value or buffer */
} PacketInfo;
```

Here, dynamic memory may be used. If the SSL packet can fit in `value` then that's where it's placed. `valueSz` holds the length and `bufferValue` is 0. If the packet is too big for `value`, only **Certificate** packets should cause this, then the packet is placed in `bufferValue`. `valueSz` still holds the size.

If memory is allocated for a **Certificate** packet then it is reclaimed after the callback returns. The timeout is implemented using signals, specifically `SIGALRM`, and is thread safe. If a previous alarm is set of type `ITIMER_REAL` then it is reset, along with the correct handler, afterwards. The old timer will be time adjusted for any time wolfSSL spends processing. If an existing timer is shorter than the passed timer, the existing timer value is used. It is still reset afterwards. An existing timer that expires will be reset if has an interval associated with it. The callback will only be issued if a timeout occurs.

See the client example for usage.

6.3 User Atomic Record Layer Processing

wolfSSL provides Atomic Record Processing callbacks for users who wish to have more control over MAC/encrypt and decrypt/verify functionality during the SSL/TLS connection.

The user will need to define 2 functions:

1. MAC/encrypt callback function
2. Decrypt/verify callback function

These two functions are prototyped by `CallbackMacEncrypt` and `CallbackDecryptVerify` in `ssl.h`:

```
typedef int (*CallbackMacEncrypt)(WOLFSSL* ssl,
    unsigned char* macOut,const unsigned char* macIn,
    unsigned int macInSz,int macContent, int macVerify,
    unsigned char* encOut, const unsigned char* encIn,
    unsigned int encSz,void* ctx);

typedef int (*CallbackDecryptVerify)(WOLFSSL* ssl,
    unsigned char* decOut, const unsigned char* decIn,
    unsigned int decSz, int content, int verify,
    unsigned int* padSz, void* ctx);
```

The user needs to write and register these functions per wolfSSL context (`WOLFSSL_CTX`) with `wolfSSL_CTX_SetMacEncryptCb()` and [wolfSSL_CTX_SetDecryptVerifyCb()](#function-wolfssl_ctx_setdecryptverifycb().

The user can set a context per WOLFSSL object (session) with `wolfSSL_SetMacEncryptCtx()` and `wolfSSL_SetDecryptVerifyCtx()`. This context may be a pointer to any user-specified context, which will then in turn be passed back to the MAC/encrypt and decrypt/verify callbacks through the `void* ctx` parameter.

1. Example callbacks can be found in `wolfssl/test.h`, under `myMacEncryptCb()` and `myDecryptVerifyCb()`. Usage can be seen in the wolfSSL example client (`examples/client/client.c`), when using the `-U` command line option.

To use Atomic Record Layer callbacks, wolfSSL needs to be compiled using the `--enable-atomicuser` configure option, or by defining the `ATOMIC_USER` preprocessor flag.

6.4 Public Key Callbacks

wolfSSL provides Public Key (PK) callbacks for users who wish to have more control over DH, ECC, Ed25519, X25519, Ed448, X448, and RSA operations during the SSL/TLS connection.

To use Public Key callbacks, wolfSSL needs to be compiled with HAVE_PK_CALLBACKS defined. This can be done using the configure option:

```
./configure --enable-pkcallbacks
```

Additionally, each callback type requires its specific feature to be enabled (e.g., HAVE_DH, HAVE_ED25519, etc.).

6.4.1 DH Callbacks

wolfSSL provides DH (Diffie-Hellman) callbacks for users who wish to have more control over DH key generation and key agreement operations during the SSL/TLS connection.

The user can optionally define 2 functions:

1. DH key generation callback
2. DH key agreement callback

These functions are prototyped by CallbackDhGenerateKeyPair and CallbackDhAgree in ssl.h:

```
typedef int (*CallbackDhGenerateKeyPair)(DhKey* key, WC_RNG* rng,
                                         byte* priv, word32* privSz,
                                         byte* pub, word32* pubSz);

typedef int (*CallbackDhAgree)(WOLFSSL* ssl, struct DhKey* key,
        const unsigned char* priv, unsigned int privSz,
        const unsigned char* otherPubKeyDer, unsigned int otherPubKeySz,
        unsigned char* out, word32* outlen,
        void* ctx);
```

The user needs to write and register these functions per wolfSSL context (WOLFSSL_CTX) with:

- wolfSSL_CTX_SetDhGenerateKeyPair()
- wolfSSL_CTX_SetDhAgreeCb()

The user can set a context per WOLFSSL object (session) with wolfSSL_SetDhAgreeCtx().

Example callbacks can be found in wolfssl/test.h, under myDhCallback(). Usage can be seen in the wolfSSL example client.

To use DH callbacks, wolfSSL needs to be compiled with HAVE_PK_CALLBACKS and HAVE_DH defined.

6.4.2 Ed25519 Callbacks

wolfSSL provides Ed25519 callbacks for users who wish to have more control over Ed25519 sign/verify operations during the SSL/TLS connection.

The user can optionally define 2 functions:

1. Ed25519 sign callback
2. Ed25519 verify callback

These functions are prototyped by CallbackEd25519Sign and CallbackEd25519Verify in ssl.h:

```
typedef int (*CallbackEd25519Sign)(WOLFSSL* ssl,
       const unsigned char* in, unsigned int inSz,
       unsigned char* out, unsigned int* outSz,
```

```
        const unsigned char* keyDer, unsigned int keySz,
        void* ctx);

typedef int (*CallbackEd25519Verify)(WOLFSSL* ssl,
        const unsigned char* sig, unsigned int sigSz,
        const unsigned char* msg, unsigned int msgSz,
        const unsigned char* keyDer, unsigned int keySz,
        int* result, void* ctx);
```

The user needs to write and register these functions per wolfSSL context (WOLFSSL_CTX) with:

- wolfSSL_CTX_SetEd25519SignCb()
- wolfSSL_CTX_SetEd25519VerifyCb()

The user can set a context per WOLFSSL object (session) with:

- wolfSSL_SetEd25519SignCtx()
- wolfSSL_SetEd25519VerifyCtx()

Example callbacks can be found in wolfssl/test.h, under myEd25519Sign() and myEd25519Verify(). Usage can be seen in the wolfSSL example client.

To use Ed25519 callbacks, wolfSSL needs to be compiled with HAVE_PK_CALLBACKS and HAVE_ED25519 defined.

6.4.3 X25519 Callbacks

wolfSSL provides X25519 callbacks for users who wish to have more control over X25519 key generation and shared secret computation during the SSL/TLS connection.

The user can optionally define 2 functions:

1. X25519 key generation callback
2. X25519 shared secret callback

These functions are prototyped by CallbackX25519KeyGen and CallbackX25519SharedSecret in ssl.h:

```
typedef int (*CallbackX25519KeyGen)(WOLFSSL* ssl, struct curve25519_key* key,
    unsigned int keySz, void* ctx);

typedef int (*CallbackX25519SharedSecret)(WOLFSSL* ssl,
        struct curve25519_key* otherKey,
        unsigned char* pubKeyDer, unsigned int* pubKeySz,
        unsigned char* out, unsigned int* outlen,
        int side, void* ctx);
```

The user needs to write and register these functions per wolfSSL context (WOLFSSL_CTX) with:

- wolfSSL_CTX_SetX25519KeyGenCb()
- wolfSSL_CTX_SetX25519SharedSecretCb()

The user can set a context per WOLFSSL object (session) with:

- wolfSSL_SetX25519KeyGenCtx()
- wolfSSL_SetX25519SharedSecretCtx()

Example callbacks can be found in wolfssl/test.h, under myX25519KeyGen() and myX25519SharedSecret(). Usage can be seen in the wolfSSL example client.

To use X25519 callbacks, wolfSSL needs to be compiled with HAVE_PK_CALLBACKS and HAVE_CURVE25519 defined.

6.4.4 Ed448 Callbacks

wolfSSL provides Ed448 callbacks for users who wish to have more control over Ed448 sign/verify operations during the SSL/TLS connection.

The user can optionally define 2 functions:

1. Ed448 sign callback
2. Ed448 verify callback

These functions are prototyped by `CallbackEd448Sign` and `CallbackEd448Verify` in `ssl.h`:

```
typedef int (*CallbackEd448Sign)(WOLFSSL* ssl,
       const unsigned char* in, unsigned int inSz,
       unsigned char* out, unsigned int* outSz,
       const unsigned char* keyDer, unsigned int keySz,
       void* ctx);

typedef int (*CallbackEd448Verify)(WOLFSSL* ssl,
       const unsigned char* sig, unsigned int sigSz,
       const unsigned char* msg, unsigned int msgSz,
       const unsigned char* keyDer, unsigned int keySz,
       int* result, void* ctx);
```

The user needs to write and register these functions per wolfSSL context (`WOLFSSL_CTX`) with:

- `wolfSSL_CTX_SetEd448SignCb()`
- `wolfSSL_CTX_SetEd448VerifyCb()`

The user can set a context per WOLFSSL object (session) with:

- `wolfSSL_SetEd448SignCtx()`
- `wolfSSL_SetEd448VerifyCtx()`

Example callbacks can be found in `wolfssl/test.h`, under `myEd448Sign()` and `myEd448Verify()`. Usage can be seen in the wolfSSL example client.

To use Ed448 callbacks, wolfSSL needs to be compiled with `HAVE_PK_CALLBACKS` and `HAVE_ED448` defined.

6.4.5 X448 Callbacks

wolfSSL provides X448 callbacks for users who wish to have more control over X448 key generation and shared secret operations during the SSL/TLS connection.

The user can optionally define 2 functions:

1. X448 key generation callback
2. X448 shared secret callback

These functions are prototyped by `CallbackX448KeyGen` and `CallbackX448SharedSecret` in `ssl.h`:

```
typedef int (*CallbackX448KeyGen)(WOLFSSL* ssl, struct curve448_key* key,
    unsigned int keySz, void* ctx);

typedef int (*CallbackX448SharedSecret)(WOLFSSL* ssl,
        struct curve448_key* otherKey,
        unsigned char* pubKeyDer, unsigned int* pubKeySz,
        unsigned char* out, unsigned int* outlen,
        int side, void* ctx);
```

The user needs to write and register these functions per wolfSSL context (WOLFSSL_CTX) with:

- wolfSSL_CTX_SetX448KeyGenCb()
- wolfSSL_CTX_SetX448SharedSecretCb()

The user can set a context per WOLFSSL object (session) with:

- wolfSSL_SetX448KeyGenCtx()
- wolfSSL_SetX448SharedSecretCtx()

Example callbacks can be found in wolfssl/test.h, under myX448KeyGen() and myX448SharedSecret(). Usage can be seen in the wolfSSL example client.

To use X448 callbacks, wolfSSL needs to be compiled with HAVE_PK_CALLBACKS and HAVE_CURVE448 defined.

6.4.6 RSA PSS Callbacks

wolfSSL provides RSA PSS callbacks for users who wish to have more control over RSA PSS sign/verify operations during the SSL/TLS connection.

The user can optionally define 4 functions:

1. RSA PSS sign callback
2. RSA PSS verify callback
3. RSA PSS sign check callback
4. RSA PSS verify check callback

These functions are prototyped by CallbackRsaPssSign, CallbackRsaPssVerify, and CallbackRsaPssSignCheck in ssl.h:

```
typedef int (*CallbackRsaPssSign)(WOLFSSL* ssl,
       const unsigned char* in, unsigned int inSz,
       unsigned char* out, unsigned int* outSz,
       int hash, int mgf,
       const unsigned char* keyDer, unsigned int keySz,
       void* ctx);

typedef int (*CallbackRsaPssVerify)(WOLFSSL* ssl,
       unsigned char* sig, unsigned int sigSz,
       unsigned char** out,
       int hash, int mgf,
       const unsigned char* keyDer, unsigned int keySz,
       void* ctx);

typedef int (*CallbackRsaPssSignCheck)(WOLFSSL* ssl,
       unsigned char* sig, unsigned int sigSz,
       unsigned char** out,
       const unsigned char* keyDer, unsigned int keySz,
       void* ctx);
```

The user needs to write and register these functions per wolfSSL context (WOLFSSL_CTX) with:

- wolfSSL_CTX_SetRsaPssSignCb()
- wolfSSL_CTX_SetRsaPssVerifyCb()
- wolfSSL_CTX_SetRsaSignCheckCb()
- wolfSSL_CTX_SetRsaPssSignCheckCb()

The user can set a context per WOLFSSL object (session) with:

- wolfSSL_SetRsaPssSignCtx()
- wolfSSL_SetRsaPssVerifyCtx()
- wolfSSL_SetRsaSignCheckCtx()
- wolfSSL_SetRsaPssSignCheckCtx()

Example callbacks can be found in wolfssl/test.h, under myRsaPssSign(), myRsaPssVerify(), and myRsaPssSignCheck(). Usage can be seen in the wolfSSL example client.

To use RSA PSS callbacks, wolfSSL needs to be compiled with HAVE_PK_CALLBACKS and WC_RSA_PSS defined.

6.4.7 ECC Callbacks

The user can optionally define 7 functions:

1. ECC sign callback
2. ECC verify callback
3. ECC shared secret callback
4. RSA sign callback
5. RSA verify callback
6. RSA encrypt callback
7. RSA decrypt callback

These two functions are prototyped by CallbackEccSign, CallbackEccVerify, CallbackEccSharedSecret, CallbackRsaSign, CallbackRsaVerify, CallbackRsaEnc, and CallbackRsaDec in ssl.h:

```
typedef int (*CallbackEccSign)(WOLFSSL* ssl, const unsigned
        char* in, unsigned int inSz, unsigned char* out,
        unsigned int* outSz, const unsigned char* keyDer,
        unsigned int keySz, void* ctx);

typedef int (*CallbackEccVerify)(WOLFSSL* ssl,
    const unsigned char* sig, unsigned int sigSz,
    const unsigned char* hash, unsigned int hashSz,
    const unsigned char* keyDer, unsigned int keySz,
    int* result, void* ctx);

typedef int (*CallbackEccSharedSecret)(WOLFSSL* ssl,
        struct ecc_key* otherKey,
    unsigned char* pubKeyDer, unsigned int* pubKeySz,
    unsigned char* out, unsigned int* outlen,
    int side, void* ctx);

typedef int (*CallbackRsaSign)(WOLFSSL* ssl,
    const unsigned char* in, unsigned int inSz,
    unsigned char* out, unsigned int* outSz,
    const unsigned char* keyDer, unsigned int keySz,
    void* ctx);

typedef int (*CallbackRsaVerify)(WOLFSSL* ssl,
    unsigned char* sig, unsigned int sigSz,
    unsigned char** out, const unsigned char* keyDer,
    unsigned int keySz, void* ctx);

typedef int (*CallbackRsaEnc)(WOLFSSL* ssl,
```

```
    const unsigned char* in, unsigned int inSz,
    unsigned char* out, unsigned int* outSz,
    const unsigned char* keyDer,
    unsigned int keySz, void* ctx);

typedef int (*CallbackRsaDec)(WOLFSSL* ssl, unsigned char* in,
    unsigned int inSz, unsigned char** out,
    const unsigned char* keyDer, unsigned int keySz,
    void* ctx);
```

The user needs to write and register these functions per wolfSSL context (WOLFSSL_CTX) with:

- wolfSSL_CTX_SetEccSignCb()
- wolfSSL_CTX_SetEccVerifyCb()
- wolfSSL_CTX_SetEccSharedSecretCb()
- wolfSSL_CTX_SetRsaSignCb()
- wolfSSL_CTX_SetRsaVerifyCb()
- wolfSSL_CTX_SetRsaEncCb()
- wolfSSL_CTX_SetRsaDecCb()

The user can set a context per WOLFSSL object (session) with:

- wolfSSL_SetEccSignCtx()
- wolfSSL_SetEccVerifyCtx()
- wolfSSL_SetEccSharedSecretCtx()
- wolfSSL_SetRsaSignCtx()
- wolfSSL_SetRsaVerifyCtx()
- wolfSSL_SetRsaEncCtx()
- wolfSSL_SetRsaDecCtx()

These contexts may be pointers to any user-specified context, which will then in turn be passed back to the respective public key callback through the void* ctx parameter.

Example callbacks can be found in wolfssl/test.h, under myEccSign(), myEccVerify(), myEccSharedSecret(), myRsaSign(), myRsaVerify(), myRsaEnc(), and myRsaDec(). Usage can be seen in the wolfSSL example client (examples/client/client.c), when using the -P command line option.

To use Atomic Record Layer callbacks, wolfSSL needs to be compiled using the --enable-pkcallbacks configure option, or by defining the HAVE_PK_CALLBACKS preprocessor flag.

6.5 Crypto Callbacks (cryptocb)

The Crypto callback framework in wolfSSL/wolfCrypt enables users to override the default implementation of select cryptographic algorithms and provide their own custom implementations at runtime. The most common use case for crypto callbacks is to offload an algorithm to custom hardware acceleration, however it could also be used to add additional logging/introspection, retrieve keys from secure storage, to call crypto from another library for standards compliance reasons, or even to perform a remote procedure call.

6.5.1 Using Crypto callbacks

The high level steps use crypto callbacks are:

1. Compile wolfSSL with crypto callback support
2. Register a callback and unique device ID (devId) with wolfCrypt
3. Pass the devId as an argument to a wolfCrypt API function
4. (TLS only): associate the devId with a wolfSSL context

6.5.1.1 1. Compile wolfSSL with crypto callback support

Support for crypto callbacks is enabled in wolfSSL via the `--enable-cryptocb` configure option, or `#define WOLF_CRYPTO_CB`.

6.5.1.2 2. Register your callback and unique devID with wolfCrypt

To use crypto callbacks, the user must register a top-level cryptographic callback function with wolfCrypt and provide a device ID (`devId`), which is an integer ID that uniquely identifies this particular callback function to wolfCrypt. This allows multiple crypto callbacks to be registered with wolfCrypt at once. The appropriate callback function will then be invoked internally by every wolfCrypt function that takes a `devId` parameter, or by wolfSSL when associated with a `WOLFSSL_CTX` or `WOLFSSL` structure.

To register a cryptographic callback function with wolfCrypt, use the `wc_CryptoCb_RegisterDevice` API. This takes a (`devId`), the callback function, and an optional user context that will be passed through to the callback when it is invoked.

```
typedef int (*CryptoDevCallbackFunc)(int devId, wc_CryptoInfo* info, void*
 ↪  ctx);

WOLFSSL_API int wc_CryptoCb_RegisterDevice(int devId,
                                           CryptoDevCallbackFunc cb,
                                           void* ctx);
```

Note: A registered `devId` can be any value other than `-2`, which is reserved for the special `INVALID_DEVID` enum value. Passing `devId == INVALID_DEVID` as an argument into a wolfCrypt API indicates that no callback should be invoked and to use the default internal implementation instead.

6.5.1.3 3. Pass the devId as an argument to a wolfCrypt API function

For wolfCrypt API's, use the init functions that accept `devId` such as:

```
wc_InitRsaKey_ex
wc_ecc_init_ex
wc_AesInit
wc_InitSha256_ex
wc_InitSha_ex
wc_HmacInit
wc_InitCmac_ex
```

This will cause wolfCrypt to invoke the crypto callback in place of the default implementation. This is not an exhaustive API list. Please refer to the wolfCrypt API documentation to see if a particular algorithm supports crypto callbacks.

6.5.1.4 4. (TLS only): associate the devId with a wolfSSL context

To enable use of a crypto callback when using TLS, you must supply the `devId` arguments on initialization of a `WOLFSSL_CTX` or `WOLFSSL` struct.

```
wolfSSL_CTX_SetDevId(ctx, devId);
wolfSSL_SetDevId(ssl, devId);
```

All relevant crypto for TLS connections associated with those structures will now invoke the crypto callback.

6.5.2 Writing your crypto callback

When a crypto callback is invoked, it will need to know what cryptographic operation has been requested, as well as the location of the input and output data and how to communicate success/failure back to wolfCrypt. This information is relayed to the callback through the `wc_CryptoInfo* info` argument.

At a high level, a crypto callback should determine if it supports the requested operation, act on the input data, update the `wc_CryptoInfo *info` argument with any relevant output data, and return a valid wolfCrypt error code.

6.5.2.1 Determining the Type of Request The `wc_CryptoInfo` structure contains a union of different structures (some of which are unions themselves), each corresponding to a specific type of cryptographic operation. The type of operation to be performed is determined by `info->algo_type`, which should be a variant of `enum wc_AlgoType` defined in `wolfssl/wolfcrypt/types.h`. Based on this `algo_type`, the appropriate union element within `wc_CryptoInfo` can be accessed to get or set parameters specific to the operation. We can refer to this top-level union as the "algo type union".

To determine the type of cryptographic request and process it accordingly, one would typically use a switch-case statement on the `algo_type` field of the `wc_CryptoInfo` structure. Each case within the switch would correspond to a different cryptographic operation, such as a symmetric cipher, hashing, public key operations, etc. There is a one-to-one mapping between `wc_AlgoType` and the corresponding algo type union, which are commented in the `wc_CryptoInfo` struct definition in `wolfssl/wolfcrypt/cryptocb.h`.

The crypto callback should return zero on success, or a valid wolfCrypt error code on failure. For unsupported algorithms, the callback should return `CRYPTOCB_UNAVAILABLE`, which will cause wolfCrypt to fall back to the internal implementation.

Here's a simplified example to illustrate this:

```
int myCryptoCallback(int devId, wc_CryptoInfo* info, void* ctx)
{
    int ret = CRYPTOCB_UNAVAILABLE;

    switch (info->algo_type) {
        case WC_ALGO_TYPE_HASH:
            /* Handle hashing, using algo type union: info->hash */
            ret = 0; /* or wolfCrypt error code */
            break;

        case WC_ALGO_TYPE_PK:
            /* Handle public key operations, using algo type union: info->pk */
            ret = 0; /* or wolfCrypt error code */
            break;

        case WC_ALGO_TYPE_CIPHER:
            /* Handle cipher operations, using algo type union: info->cipher */
            ret = 0; /* or wolfCrypt error code */
            break;

        /* and so on for other algo types... */
    }

    return ret; // Return success or an appropriate error code
}
```

Some of the simpler algo type unions, such as the RNG and seed unions, require no further processing and can be immediately acted on. However, more complicated operations like cipher or public key types have multiple union variants unto themselves that must be conditionally interpreted in the same manner. The variants and levels of hierarchy of each union are specific to each category of algorithm type, the full definitions of which can be found in the definition of the `wc_CryptoInfo` structure. As a general rule, each level that is a union will contain some sort of type indicator informing how the rest

of the union should be dereferenced.

Here is a simplified example for a complex algo type union with multiple levels of union decoding. The callback contains support for random number generation, as well as ECC key agreement, sign, and verify.

```
int myCryptoCallback(int devId, wc_CryptoInfo* info, void* ctx)
{
    int ret = CRYPTOCB_UNAVAILABLE;

    switch (info->algo_type) {
        case WC_ALGO_TYPE_PK:
            /* Handle public key operations, using algo type union: info->pk */
            switch (info->pk.type) { /* pk type is of type wc_PkType */
                case WC_PK_TYPE_ECDH:
                    /* use info->pk.ecdh */
                    ret = 0; /* or wolfCrypt error code */
                    break;

                case WC_PK_TYPE_ECDSA_SIGN:
                    /* use info->pk.eccsign */
                    ret = 0; /* or wolfCrypt error code */
                    break;

                case WC_PK_TYPE_ECDSA_VERIFY:
                    /* use info->pk.eccverify */
                    ret = 0; /* or wolfCrypt error code */
                    break;
            }
            break;

        case WC_ALGO_TYPE_RNG:
            /* use info->rng */
            ret = 0; /* or wolfCrypt error code */
            break;
    }

    return ret;
}
```

6.5.3 Handling the request

The data structure for each type of request generally contains a pointer and associated size for input and output data. Depending on the request it may include additional data including cryptographic keys, nonces, or further configuration. The crypto callback should operate on the input data and write relevant output data back to the appropriate variant of the algo type union. Writing data to a union variant that does not correspond to the algorithm type in question (e.g. using `info->cipher` when `info->algo_type == WC_ALGO_TYPE_RNG` is a memory error and can result in undefined behavior.

6.5.4 Troubleshooting

Some older compilers don't allow "anonymous inline aggregates", which wolfCrypt uses when defining the nested `wc_CryptoInfo` anonymous unions in order to save space. To disable the use of anonymous inline aggregates, define the `HAVE_ANONYMOUS_INLINE_AGGREGATES` preprocessor macro as `0`.

This will allow crypto callbacks to be used, but will dramatically increase the size of the `wc_CryptoInfo` structure.

6.5.5 Examples

Full examples of crypto callbacks can be found in the following locations

- VaultIC Crypto Callbacks
- STSAFE-A100 ECC Crypto Callbacks
- TPM 2.0 wolfTPM Crypto Callbacks
- Generic wolfCrypt tests

COPYRIGHT ©2024 wolfSSL Inc.

7 Keys and Certificates

For an introduction to X.509 certificates, as well as how they are used in SSL and TLS, please see Appendix A.

7.1 Supported Formats and Sizes

wolfSSL (formerly CyaSSL) has support for **PEM**, and **DER** formats for certificates and keys, as well as PKCS#8 private keys (with PKCS#5 or PKCS#12 encryption).

PEM, or "Privacy Enhanced Mail" is the most common format that certificates are issued in by certificate authorities. PEM files are Base64 encoded ASCII files which can include multiple server certificates, intermediate certificates, and private keys, and usually have a `.pem`, `.crt`, `.cer`, or `.key` file extension. Certificates inside PEM files are wrapped in the "`-----BEGIN CERTIFICATE-----`" and "`-----END CERTIFICATE-----`" statements.

DER, or "Distinguished Encoding Rules", is a binary format of a certificate. DER file extensions can include `.der` and `.cer`, and cannot be viewed with a text editor.

An X.509 certificate is encoded using ASN.1 format. The DER format is the ASN.1 encoding. The PEM format is Base64 encoded and wrapped with a human readable header and footer. TLS sends certificates in DER format.

PKCS #12, also called PFX, is a password-protected bundle that carries a private key together with its certificate and CA chain in a single file, usually with a `.p12` or `.pfx` extension. wolfSSL can read and write these bundles through the wolfCrypt PKCS #12 API; see PKCS #12 for details.

7.2 Supported Certificate Extensions

If an unsupported or unknown extension that is marked as critical is found, then an error message is returned, otherwise unsupported or unknown extensions found are ignored. Certificate extension parsing expects that at the very least `--enable-certext` (macro WOLFSSL_CERT_EXT) has been used when compiling the wolfSSL library. This is a high level list of certificate extensions that can be **parsed** and at least part, if not all, of the extensions be used.

Extension From RFC 5280	Supported
Authority Key Identifier	Yes
Subject Key Identifier	Yes
Key Usage	Yes
Certificate Policies	Yes
Policy Mappings	No
Subject Alternative Name	Yes
Issuer Alternative Name	No
Subject Directory Attributes	No
Basic Constraints	Yes
Name Constraints	Yes
Policy Constraints	Yes
Extended Key Usage	Yes
CRL Distribution Points	Yes
Inihibit anyPolicy	Yes
Freshest CRL	No

COPYRIGHT ©2024 wolfSSL Inc.

Additional Extension	Supported
Netscape	Yes
Custom OID	Yes

The next couple of sections delve deeper into the support of the individual certificate extensions.

7.2.0.1 Auth/Subject Key ID By default key ID's are a SHA1 hash of the key. SHA256 hashes of the key are also supported.

7.2.0.2 Subject Alternative Names

Alternative Name Types	Supported
email	Yes
DNS name	Yes
IP address	Yes
URI	Yes

7.2.0.3 Key Usage Key usage can be parsed and retrieved after parsing of the certificate is complete.

Key Usage	Supported
digitalSignature	Yes
nonRepudiation	Yes
keyEncipherment	Yes
dataEncipherment	Yes
keyAgreement	Yes
keyCertSign	Yes
cRLSign	Yes
encipherOnly	Yes
decipherOnly	Yes

7.2.0.4 Extended Key Usage Starting at wolfSSL version 3.15.5 if an extended key usage is unknown/unsupported then it is ignored. For versions before 3.15.5 an unknown extended key usage OID will cause a parsing error.

Extended Key Usage	Supported
anyExtendedKeyUsage	Yes
id-kp-serverAuth	Yes
id-kp-clientAuth	Yes
id-kp-codeSigning	Yes
id-kp-emailProtection	Yes
id-kp-timeStamping	Yes
id-kp-OCSPSigning	Yes

COPYRIGHT ©2024 wolfSSL Inc.

7.2.0.5 Custom OID Extensions Custom OID X.509 extension injection and parsing was introduced in wolfSSL version 5.3.0. The enable options `--enable-certgen` must be used along with the macros `WOLFSSL_ASN_TEMPLATE`, `WOLFSSL_CUSTOM_OID` and `HAVE_OID_ENCODING` to enable custom extensions. `WOLFSSL_ASN_TEMPLATE` is defined by default when using `./configure` but needs to be manually defined if building with `WOLFSSL_USER_SETTINGS`.

After building wolfSSL with these settings the function `wc_SetCustomExtension()` can be used to set a custom extension in a `Cert` struct and `wc_SetUnknownExtCallback()` can be used to register a callback for handling unknown extension OIDs in a `DecodedCert` struct.

7.3 Certificate Loading

Certificates are normally loaded using the file system (although loading from memory buffers is supported as well - see No File System and using Certificates).

7.3.1 Loading CA Certificates**

CA certificate files can be loaded using the `wolfSSL_CTX_load_verify_locations()` function:

```
int wolfSSL_CTX_load_verify_locations(WOLFSSL_CTX *ctx,
                                      const char *CAfile,
                                      const char *CApath);
```

CA loading can also parse multiple CA certificates per file using the above function by passing in a `CAfile` in PEM format with as many certs as possible. This makes initialization easier, and is useful when a client needs to load several root CAs at startup. This makes wolfSSL easier to port into tools that expect to be able to use a single file for CAs.

NOTE: If you have to load a chain of Roots and Intermediate certificates you must load them in the order of trust. Load ROOT CA first followed by Intermediate 1 followed by Intermediate 2 and so on. You may call `wolfSSL_CTX_load_verify_locations()` for each cert to be loaded or just once with a file containing the certs in order (Root at the top of the file and certs ordered by the chain of trust)

7.3.2 Loading Client or Server Certificates

Loading single client or server certificates can be done with the `wolfSSL_CTX_use_certificate_file()` function. If this function is used with a certificate chain, only the actual, or "bottom" certificate will be sent.

```
int wolfSSL_CTX_use_certificate_file(WOLFSSL_CTX *ctx,
                                     const char *CAfile,
                                     int type);
```

`CAfile` is the CA certificate file, and `type` is the format of the certificate - such as `SSL_FILETYPE_PEM`.

The server and client can send certificate chains using the `wolfSSL_CTX_use_certificate_chain_file()` function. The certificate chain file must be in PEM format and must be sorted starting with the subject's certificate (the actual client or server cert), followed by any intermediate certificates and ending (optionally) at the root "top" CA. The example server (`/examples/server/server.c`) uses this functionality.

NOTE: This is the exact reverse of the order necessary when loading a certificate chain for verification! Your file contents in this scenario would be Entity cert at the top of the file followed by the next cert up the chain and so on with Root CA at the bottom of the file.

```
int wolfSSL_CTX_use_certificate_chain_file(WOLFSSL_CTX *ctx,
                                   const char *file);
```

7.3.3 Loading Private Keys

Server private keys can be loaded using the wolfSSL_CTX_use_PrivateKey_file() function.

```
int wolfSSL_CTX_use_PrivateKey_file(WOLFSSL_CTX *ctx,
                      const char *keyFile, int type);
```

keyFile is the private key file, and type is the format of the private key (e.g. SSL_FILETYPE_PEM).

7.3.4 Loading Trusted Peer Certificates

Loading a trusted peer certificate to use can be done with wolfSSL_CTX_trust_peer_cert().

```
int wolfSSL_CTX_trust_peer_cert(WOLFSSL_CTX *ctx,
                 const char *trustCert, int type);
```

trustCert is the certificate file to load, and type is the format of the private key (i.e. SSL_FILETYPE_PEM).

7.4 Certificate Chain Verification

wolfSSL requires only the top or "root" certificate in a chain to be loaded as a trusted certificate in order to verify a certificate chain. This means that if you have a certificate chain (A -> B -> C), where C is signed by B, and B is signed by A, wolfSSL only requires that certificate A be loaded as a trusted certificate in order to verify the entire chain (A->B->C).

For example, if a server certificate chain looks like:

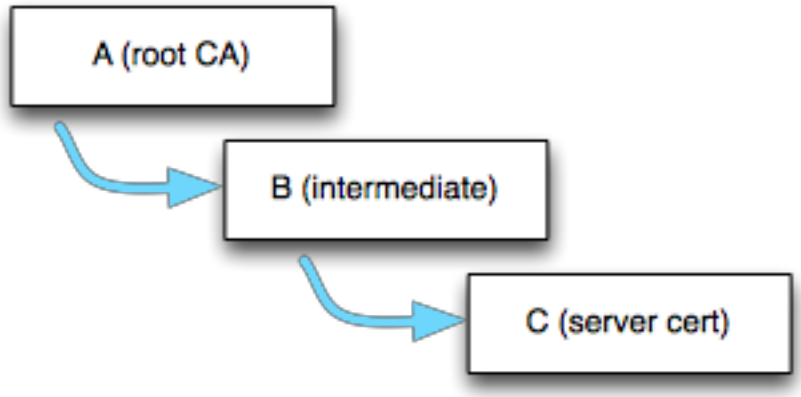

Figure 4: Certificate Chain

The wolfSSL client should already have at least the root cert (A) loaded as a trusted root (with wolfSSL_CTX_load_verify_locations()). When the client receives the server cert chain, it uses the signature of A to verify B, and if B has not been previously loaded into wolfSSL as a trusted root, B gets stored in wolfSSL's internal cert chain (wolfSSL just stores what is necessary to verify a certificate: common name hash, public key and key type, etc.). If B is valid, then it is used to verify C.

Following this model, as long as root cert "A" has been loaded as a trusted root into the wolfSSL server, the server certificate chain will still be able to be verified if the server sends (A->B->C), or (B->C). If the server just sends (C), and not the intermediate certificate, the chain will not be able to be verified unless the wolfSSL client has already loaded B as a trusted root.

7.5 Domain Name Check for Server Certificates

wolfSSL has an extension on the client that automatically checks the domain of the server certificate. In OpenSSL mode nearly a dozen function calls are needed to perform this. wolfSSL checks that the date of the certificate is in range, verifies the signature, and additionally verifies the domain if you call `wolfSSL_check_domain_name(WOLFSSL* ssl, const char* dn)` before calling `wolfSSL_connect()`. wolfSSL will match the X.509 subject name of peer's server certificate against dn (the expected domain name). If the names match `wolfSSL_connect()` will proceed normally, however if there is a name mismatch, `wolfSSL_connect()` will return a fatal error and `wolfSSL_get_error()` will return `DOMAIN_NAME_MISMATCH`.

Checking the domain name of the certificate is an important step that verifies the server is actually who it claims to be. This extension is intended to ease the burden of performing the check.

7.6 No File System and using Certificates

Normally a file system is used to load private keys, certificates, and CAs. Since wolfSSL is sometimes used in environments without a full file system, an extension to use memory buffers instead is provided. To use the extension define the constant `NO_FILESYSTEM` and the following functions will be made available:

- `int wolfSSL_CTX_load_verify_buffer(WOLFSSL_CTX* ctx, const unsigned char* in,long sz, int format);`
- `int wolfSSL_CTX_use_certificate_buffer(WOLFSSL_CTX* ctx, const unsigned char* in, long sz, int format);`
- `int wolfSSL_CTX_use_PrivateKey_buffer(WOLFSSL_CTX* ctx, const unsigned char* in, long sz, int format);`
- `int wolfSSL_CTX_use_certificate_chain_buffer(WOLFSSL_CTX* ctx, const unsigned char* in,long sz);`
- `int wolfSSL_CTX_trust_peer_buffer(WOLFSSL_CTX* ctx, const unsigned char* in, Long sz, int format);`

Use these functions exactly like their counterparts that are named `*_file` instead of `*_buffer`. And instead of providing a filename provide a memory buffer. See API documentation for usage details.

7.6.1 Test Certificate and Key Buffers

wolfSSL comes bundled with test certificate and key files. Additionally, it also comes bundled with test certificate and key buffers for use in environments with no filesystem available. These buffers are available in `certs_test.h` when defining one or more of `USE_CERT_BUFFERS_1024`, `USE_CERT_BUFFERS_2048`, or `USE_CERT_BUFFERS_256`.

7.7 Serial Number Retrieval

The serial number of an X.509 certificate can be extracted from wolfSSL using `wolfSSL_X509_get_serial_number()`. The serial number can be of any length.

```
int wolfSSL_X509_get_serial_number(WOLFSSL_X509* x509,
            unsigned char* buffer, int* inOutSz)
```

`buffer` will be written to with at most `*inOutSz` bytes on input. After the call, if successful (return of 0), `*inOutSz` will hold the actual number of bytes written to `buffer`. A full example is included `wolfssl/test.h`.

7.8 RSA Key Generation

wolfSSL supports RSA key generation of varying lengths up to 4096 bits. Key generation is off by default but can be turned on during the `./configure` process with `--enable-keygen` or by defining `WOLFSSL_KEY_GEN` in Windows or non-standard environments. Creating a key is easy, only requiring one function from `rsa.h`:

```
int wc_MakeRsaKey(RsaKey* key, int size, long e, RNG* rng);
```

Where `size` is the length in bits and e is the public exponent (using 65537 is usually a good choice for e). The following from `wolfcrypt/test/test.c` gives an example creating an RSA key of 1024 bits:

```
RsaKey genKey;
RNG    rng;
int    ret;

wc_InitRng(&rng);
wc_InitRsaKey(&genKey, 0);

ret = wc_MakeRsaKey(&genKey, 1024, 65537, &rng);
if (ret != 0)
    /* ret contains error */;
```

The RsaKey genKey can now be used like any other RsaKey. If you need to export the key, wolfSSL provides both DER and PEM formatting in asn.h. Always convert the key to DER format first, and then if you need PEM use the generic `wc_DerToPem()` function like this:

```
byte der[4096];
int  derSz = wc_RsaKeyToDer(&genKey, der, sizeof(der));
if (derSz < 0)
    /* derSz contains error */;
```

The buffer `der` now holds a DER format of the key. To convert the DER buffer to PEM use the conversion function:

```
byte pem[4096];
int  pemSz = wc_DerToPem(der, derSz, pem, sizeof(pem),
                         PRIVATEKEY_TYPE);
if (pemSz < 0)
    /* pemSz contains error */;
```

The last argument of *DerToPem()* takes a type parameter, usually either `PRIVATEKEY_TYPE` or `CERT_TYPE`. Now the buffer `pem` holds the PEM format of the key. Supported types are:

- `CA_TYPE`
- `TRUSTED_PEER_TYPE`
- `CERT_TYPE`
- `CRL_TYPE`
- `DH_PARAM_TYPE`
- `DSA_PARAM_TYPE`
- `CERTREQ_TYPE`
- `DSA_TYPE`
- `DSA_PRIVATEKEY_TYPE`
- `ECC_TYPE`
- `ECC_PRIVATEKEY_TYPE`
- `RSA_TYPE`
- `PRIVATEKEY_TYPE`
- `ED25519_TYPE`
- `EDDSA_PRIVATEKEY_TYPE`

- PUBLICKEY_TYPE
- ECC_PUBLICKEY_TYPE
- PKCS8_PRIVATEKEY_TYPE
- PKCS8_ENC_PRIVATEKEY_TYPE

7.8.1 RSA Key Generation Notes

The RSA private key contains the public key as well. The private key can be used as both a private and public key by wolfSSL as used in test.c. The private key and the public key (in the form of a certificate) is all that is typically needed for SSL.

A separate public key can be loaded into wolfSSL manually using the wc_RsaPublicKeyDecode() function if need be. Additionally, the `wc_RsaKeyToPublicDer()` function can be used to export the public RSA key.

7.9 Certificate Generation

wolfSSL supports X.509 v3 certificate generation. Certificate generation is off by default but can be turned on during the `./configure` process with `--enable-certgen` or by defining `WOLFSSL_CERT_GEN` in Windows or non-standard environments.

Before a certificate can be generated the user needs to provide information about the subject of the certificate. This information is contained in a structure from `wolfssl/wolfcrypt/asn_public.h` named `Cert`:

```
/* for user to fill for certificate generation */
typedef struct Cert {
    int      version;                   /* x509 version  */
    byte     serial[CTC_SERIAL_SIZE];   /* serial number */
    int      sigType;                   /*signature algo type */
    CertName issuer;                    /* issuer info */
    int      daysValid;                 /* validity days */
    int      selfSigned;                /* self signed flag */
    CertName subject;                   /* subject info */
    int      isCA;                      /*is this going to be a CA*/
    ...
} Cert;
```

Where CertName looks like:

```
typedef struct CertName {
char country[CTC_NAME_SIZE];
    char countryEnc;
    char state[CTC_NAME_SIZE];
    char stateEnc;
    char locality[CTC_NAME_SIZE];
    char localityEnc;
    char sur[CTC_NAME_SIZE];
    char surEnc;
    char org[CTC_NAME_SIZE];
    char orgEnc;
    char unit[CTC_NAME_SIZE];
    char unitEnc;
    char commonName[CTC_NAME_SIZE];
    char commonNameEnc;
```

```
    char email[CTC_NAME_SIZE];  /* !!!! email has to be last!!!! */
} CertName;
```

Before filling in the subject information an initialization function needs to be called like this:

```
Cert myCert;
wc_InitCert(&myCert);
```

`InitCert()` sets defaults for some of the variables including setting the version to **3** (0x02), the serial number to **0** (randomly generated), the sigType to `CTC_SHAwRSA`, the daysValid to **500**, and selfSigned to **1** (TRUE). Supported signature types include:

- `CTC_SHAwDSA`
- `CTC_MD2wRSA`
- `CTC_MD5wRSA`
- `CTC_SHAwRSA`
- `CTC_SHAwECDSA`
- `CTC_SHA256wRSA`
- `CTC_SHA256wECDSA`
- `CTC_SHA384wRSA`
- `CTC_SHA384wECDSA`
- `CTC_SHA512wRSA`
- `CTC_SHA512wECDSA`

Now the user can initialize the subject information like this example from `wolfcrypt/test/test.c`:

```
strncpy(myCert.subject.country, "US", CTC_NAME_SIZE);
strncpy(myCert.subject.state, "OR", CTC_NAME_SIZE);
strncpy(myCert.subject.locality, "Portland", CTC_NAME_SIZE);
strncpy(myCert.subject.org, "yaSSL", CTC_NAME_SIZE);
strncpy(myCert.subject.unit, "Development", CTC_NAME_SIZE);
strncpy(myCert.subject.commonName, "www.wolfssl.com", CTC_NAME_SIZE);
strncpy(myCert.subject.email, "facts@wolfssl.com", CTC_NAME_SIZE);
```

Then, a self-signed certificate can be generated using the variables genKey and rng from the above key generation example (of course any valid RsaKey or RNG can be used):

```
byte derCert[4096];

int certSz = wc_MakeSelfCert(&myCert, derCert, sizeof(derCert), &key, &rng);
if (certSz < 0)
  /* certSz contains the error */;
```

The buffer `derCert` now contains a DER format of the certificate. If you need a PEM format of the certificate you can use the generic `wc_DerToPem()` function and specify the type to be `CERT_TYPE` like this:

```
byte* pemCert[4096];

int pemCertSz = wc_DerToPem(derCert, certSz, pemCert, sizeof(pemCert),
 ↪  CERT_TYPE);
if (pemCertSz < 0)
  /* pemCertSz contains error */;
```

Supported types are:

- `CA_TYPE`
- `TRUSTED_PEER_TYPE`
- `CERT_TYPE`
- `CRL_TYPE`

- DH_PARAM_TYPE
- DSA_PARAM_TYPE
- CERTREQ_TYPE
- DSA_TYPE
- DSA_PRIVATEKEY_TYPE
- ECC_TYPE
- ECC_PRIVATEKEY_TYPE
- RSA_TYPE
- PRIVATEKEY_TYPE
- ED25519_TYPE
- EDDSA_PRIVATEKEY_TYPE
- PUBLICKEY_TYPE
- ECC_PUBLICKEY_TYPE
- PKCS8_PRIVATEKEY_TYPE
- PKCS8_ENC_PRIVATEKEY_TYPE

Now the buffer pemCert holds the PEM format of the certificate.

If you wish to create a CA signed certificate then a couple of steps are required. After filling in the subject information as before, you'll need to set the issuer information from the CA certificate. This can be done with wc_SetIssuer() like this:

```
ret = wc_SetIssuer(&myCert, "ca-cert.pem");
if (ret < 0)
    /* ret contains error */;
```

Then you'll need to perform the two-step process of creating the certificate and then signing it (wc_MakeSelfCert() does these both in one step). You'll need the private keys from both the issuer (caKey) and the subject (key). Please see the example in test.c for complete usage.

```
byte derCert[4096];

int certSz = wc_MakeCert(&myCert, derCert, sizeof(derCert), &key, NULL, &rng);
if (certSz < 0);
   /*certSz contains the error*/;

certSz = wc_SignCert(myCert.bodySz, myCert.sigType, derCert,
            sizeof(derCert), &caKey, NULL, &rng);
if (certSz < 0);
   /*certSz contains the error*/;
```

The buffer derCert now contains a DER format of the CA signed certificate. If you need a PEM format of the certificate please see the self signed example above. Note that MakeCert() and SignCert() provide function parameters for either an RSA or ECC key to be used. The above example uses an RSA key and passes NULL for the ECC key parameter.

7.10 Certificate Signing Request (CSR) Generation

wolfSSL supports X.509 v3 certificate signing request (CSR) generation. CSR generation is off by default but can be turned on during the ./configure process with --enable-certreq --enable-certgen or by defining WOLFSSL_CERT_GEN **and** WOLFSSL_CERT_REQ in Windows or non-standard environments.

Before a CSR can be generated the user needs to provide information about the subject of the certificate. This information is contained in a structure from wolfssl/wolfcrypt/asn_public.h named Cert:

For details on the Cert and CertName structures please reference Certificate Generation above.

Before filling in the subject information an initialization function needs to be called like this:

```
Cert request;
wc_InitCert(&request);
```

`InitCert()` sets defaults for some of the variables including setting the version to **3** (0x02), the serial number to **0** (randomly generated), the sigType to CTC_SHAwRSA, the daysValid to **500**, and selfSigned to **1** (TRUE). Supported signature types include:

- CTC_SHAwDSA
- CTC_MD2wRSA
- CTC_MD5wRSA
- CTC_SHAwRSA
- CTC_SHAwECDSA
- CTC_SHA256wRSA
- CTC_SHA256wECDSA
- CTC_SHA384wRSA
- CTC_SHA384wECDSA
- CTC_SHA512wRSA
- CTC_SHA512wECDSA

Now the user can initialize the subject information like this example from https://github.com/wolfSSL/wolfssl-examples/blob/master/certgen/csr_example.c:

```
strncpy(req.subject.country, "US", CTC_NAME_SIZE);
strncpy(req.subject.state, "OR", CTC_NAME_SIZE);
strncpy(req.subject.locality, "Portland", CTC_NAME_SIZE);
strncpy(req.subject.org, "wolfSSL", CTC_NAME_SIZE);
strncpy(req.subject.unit, "Development", CTC_NAME_SIZE);
strncpy(req.subject.commonName, "www.wolfssl.com", CTC_NAME_SIZE);
strncpy(req.subject.email, "facts@wolfssl.com", CTC_NAME_SIZE);
```

Then, a valid signed CSR can be generated using the variable key from the above key generation example (of course any valid ECC/RSA key or RNG can be used):

```
byte der[4096]; /* Store request in der format once made */

ret = wc_MakeCertReq(&request, der, sizeof(der), NULL, &key);
/* check ret value for error handling, <= 0 indicates a failure */
```

Next you will want to sign your request making it valid, use the rng variable from the above key generation example. (of course any valid ECC/RSA key or RNG can be used)

```
derSz = ret;

req.sigType = CTC_SHA256wECDSA;
ret = wc_SignCert(request.bodySz, request.sigType, der, sizeof(der), NULL,
 ↪  &key, &rng);
/* check ret value for error handling, <= 0 indicates a failure */
```

Lastly it is time to convert the CSR to PEM format for sending to a CA authority to use in issuing a certificate:

```
ret = wc_DerToPem(der, derSz, pem, sizeof(pem), CERTREQ_TYPE);
/* check ret value for error handling, <= 0 indicates a failure */
printf("%s", pem); /* or write to a file */
```

7.10.1 Limitations

There are fields that are mandatory in a certificate that are excluded in a CSR. There are other fields in a CSR that are also deemed "optional" that are otherwise mandatory when in a certificate. Because of this the wolfSSL certificate parsing engine, which strictly checks all certificate fields AND considers all fields mandatory, does not support consuming a CSR at this time. Therefore while CSR generation AND certificate generation from scratch are supported, wolfSSL does not support certificate generation FROM a CSR. Passing in a CSR to the wolfSSL parsing engine will return a failure at this time. Check back for updates once we support consuming a CSR for use in certificate generation!

See also: Certificate Generation

7.11 Convert to raw ECC key

With our recently added support for raw ECC key import comes the ability to convert an ECC key from PEM to DER. Use the following with the specified arguments to accomplish this:

```
wc_EccKeyToDer(ecc_key*, byte* output, word32 inLen);
```

7.11.1 Example

```
#define FOURK_BUF 4096
byte  der[FOURK_BUF];
ecc_key userB;

wc_EccKeyToDer(&userB, der, FOURK_BUF);
```

8 Debugging

8.1 Debugging and Logging

wolfSSL (formerly CyaSSL) has support for debugging through log messages in environments where debugging is limited. To turn logging on use the function `wolfSSL_Debugging_ON()` and to turn it off use `wolfSSL_Debugging_OFF()`. In a normal build (release mode) these functions will have no effect. In a debug build, define `DEBUG_WOLFSSL` to ensure these functions are turned on.

As of wolfSSL 2.0, logging callback functions may be registered at runtime to provide more flexibility with how logging is done. The logging callback can be registered with the function `wolfSSL_SetLoggingCb()`:

```
int wolfSSL_SetLoggingCb(wolfSSL_Logging_cb log_function);

typedef void (*wolfSSL_Logging_cb)(const int logLevel,
                  const char *const logMessage);
```

The log levels can be found in `wolfssl/wolfcrypt/logging.h`, and the implementation is located in `logging.c`. By default, wolfSSL logs to `stderr` with `fprintf`.

8.2 Error Codes

wolfSSL tries to provide informative error messages in order to help with debugging.

Each `wolfSSL_read()` and `wolfSSL_write()` call will return the number of bytes written upon success, 0 upon connection closure, and -1 for an error, just like `read()` and `write()`. In the event of an error you can use two calls to get more information about the error.

The function `wolfSSL_get_error()` will return the current error code. It takes the current `WOLFSSL` object, and `wolfSSL_read()` or `wolfSSL_write()` result value as arguments and returns the current error code.

```
int err = wolfSSL_get_error(ssl, result);
```

To get a more human-readable error code description, the `wolfSSL_ERR_error_string()` function can be used. It takes the return code from `wolfSSL_get_error` and a storage buffer as arguments, and places the corresponding error description into the storage buffer (`errorString` in the example below).

```
char errorString[80];
wolfSSL_ERR_error_string(err, errorString);
```

If you are using non blocking sockets, you can test for errno EAGAIN/EWOULDBLOCK or more correctly you can test the specific error code for `SSL_ERROR_WANT_READ` or `SSL_ERROR_WANT_WRITE`.

For a list of wolfSSL and wolfCrypt error codes, please see Appendix C (Error Codes).

9 Library Design

9.1 Library Headers

With the release of wolfSSL 2.0.0 RC3, library header files are now located in the following locations:

- wolfSSL: `wolfssl/`
- wolfCrypt: `wolfssl/wolfcrypt/`
- wolfSSL OpenSSL Compatibility Layer: `wolfssl/openssl/`

When using the OpenSSL Compatibility layer (see OpenSSL Compatibility), the `/wolfssl/openssl/ssl.h` header is required to be included:

```
#include <wolfssl/openssl/ssl.h>
```

When using only the wolfSSL native API, only the `/wolfssl/ssl.h` header is required to be included:

```
#include <wolfssl/ssl.h>
```

9.2 Startup and Exit

All applications should call `wolfSSL_Init()` before using the library and call `wolfSSL_Cleanup()` at program termination. Currently these functions only initialize and free the shared mutex for the session cache in multi-user mode but in the future they may do more so it's always a good idea to use them.

9.3 Structure Usage

In addition to header file location changes, the release of wolfSSL 2.0.0 RC3 created a more visible distinction between the native wolfSSL API and the wolfSSL OpenSSL Compatibility Layer. With this distinction, the main SSL/TLS structures used by the native wolfSSL API have changed names. The new structures are as follows. The previous names are still used when using the OpenSSL Compatibility Layer (see OpenSSL Compatibility).

- `WOLFSSL` (previously SSL)
- `WOLFSSL_CTX` (previously SSL_CTX)
- `WOLFSSL_METHOD` (previously SSL_METHOD)
- `WOLFSSL_SESSION` (previously SSL_SESSION)
- `WOLFSSL_X509` (previously X509)
- `WOLFSSL_X509_NAME` (previously X509_NAME)
- `WOLFSSL_X509_CHAIN` (previously X509_CHAIN)

9.4 Thread Safety

wolfSSL (formerly CyaSSL) is thread safe by design. Multiple threads can enter the library simultaneously without creating conflicts because wolfSSL avoids global data, static data, and the sharing of objects. The user must still take care to avoid potential problems in some areas.

1. A client may share an WOLFSSL object across multiple threads but access must be synchronized, i.e., trying to read/write at the same time from two different threads with the same SSL pointer is not supported.

   wolfSSL could take a more aggressive (constrictive) stance and lock out other users when a function is entered that cannot be shared but this level of granularity seems counter-intuitive. All users (even single threaded ones) will pay for the locking and multi-thread ones won't be able to re-enter the library even if they aren't sharing objects across threads. This penalty seems much

too high and wolfSSL leaves the responsibility of synchronizing shared objects in the hands of the user.

2. Besides sharing WOLFSSL pointers, users must also take care to completely initialize an WOLFSSL_CTX before passing the structure to wolfSSL_new(). The same WOLFSSL_CTX can create multiple WOLFSSL structs but the WOLFSSL_CTX is only read during wolfSSL_new() creation and any future (or simultaneous changes) to the WOLFSSL_CTX will not be reflected once the WOLFSSL object is created.

   Again, multiple threads should synchronize writing access to a WOLFSSL_CTX and it is advised that a single thread initialize the WOLFSSL_CTX to avoid the synchronization and update problem described above.

3. Some optimizations allocate memory on a per thread basis. If fixed point ECC cache is enabled (FP_ECC), then threads should release the cached buffers using wc_ecc_fp_free() before the thread exits.

9.5 Input and Output Buffers

wolfSSL now uses dynamic buffers for input and output. They default to 0 bytes and are controlled by the RECORD_SIZE define in wolfssl/internal.h. If an input record is received that is greater in size than the static buffer, then a dynamic buffer is temporarily used to handle the request and then freed. You can set the static buffer size up to the MAX_RECORD_SIZE which is 2^14 or 16,384.

If you prefer the previous way that wolfSSL operated, with 16Kb static buffers that will never need dynamic memory, you can still get that option by defining LARGE_STATIC_BUFFERS.

If dynamic buffers are used and the user requests a wolfSSL_write() that is bigger than the buffer size, then a dynamic block up to MAX_RECORD_SIZE is used to send the data. Users wishing to only send the data in chunks of at most RECORD_SIZE size can do this by defining STATIC_CHUNKS_ONLY. This will cause wolfSSL to use I/O buffers which grow up to RECORD_SIZE, which is 128 bytes by default.

10 wolfCrypt Usage Reference

wolfCrypt is the cryptography library primarily used by wolfSSL. It is optimized for speed, small footprint, and portability. wolfSSL interchanges with other cryptography libraries as required.

Types used in the examples:

```
typedef unsigned char byte;
typedef unsigned int  word32;
```

10.1 Hash Functions

10.1.1 MD4

NOTE: MD4 is outdated and considered insecure. Please consider using a different hashing function if possible.

To use MD4 include the MD4 header `wolfssl/wolfcrypt/md4.h`. The structure to use is `Md4`, which is a typedef. Before using, the hash initialization must be done with the `wc_InitMd4()` call. Use `wc_Md4Update()` to update the hash and `wc_Md4Final()` to retrieve the final hash.

```
byte md4sum[MD4_DIGEST_SIZE];
byte buffer[1024];
/* fill buffer with data to hash*/

Md4 md4;
wc_InitMd4(&md4);

wc_Md4Update(&md4, buffer, sizeof(buffer));  /*can be called again
                                    and again*/
wc_Md4Final(&md4, md4sum);
```

`md4sum` now contains the digest of the hashed data in buffer.

10.1.2 MD5

NOTE: MD5 is outdated and considered insecure. Please consider using a different hashing function if possible.

To use MD5 include the MD5 header `wolfssl/wolfcrypt/md5.h`. The structure to use is `Md5`, which is a typedef. Before using, the hash initialization must be done with the `wc_InitMd5()` call. Use `wc_Md5Update()` to update the hash and `wc_Md5Final()` to retrieve the final hash

```
byte md5sum[MD5_DIGEST_SIZE];
byte buffer[1024];
/*fill buffer with data to hash*/

Md5 md5;
wc_InitMd5(&md5);

wc_Md5Update(&md5, buffer, sizeof(buffer));  /*can be called again
                                    and again*/
wc_Md5Final(&md5, md5sum);
```

`md5sum` now contains the digest of the hashed data in buffer.

10.1.3 SHA / SHA-224 / SHA-256 / SHA-384 / SHA-512

To use SHA include the SHA header `wolfssl/wolfcrypt/sha.h`. The structure to use is Sha, which is a typedef. Before using, the hash initialization must be done with the `wc_InitSha()` call. Use `wc_ShaUpdate()` to update the hash and `wc_ShaFinal()` to retrieve the final hash:

```
byte shaSum[SHA_DIGEST_SIZE];
byte buffer[1024];
/*fill buffer with data to hash*/

Sha sha;
wc_InitSha(&sha);

wc_ShaUpdate(&sha, buffer, sizeof(buffer));  /*can be called again
                                               and again*/
wc_ShaFinal(&sha, shaSum);
```

`shaSum` now contains the digest of the hashed data in buffer.

To use either SHA-224, SHA-256, SHA-384, or SHA-512, follow the same steps as shown above, but use either the `wolfssl/wolfcrypt/sha256.h` or `wolfssl/wolfcrypt/sha512.h` (for both SHA-384 and SHA-512). The SHA-256, SHA-384, and SHA-512 functions are named similarly to the SHA functions.

For **SHA-224**, the functions `wc_InitSha224()`, `wc_Sha224Update()`, and `wc_Sha224Final()` will be used with the structure Sha224.

For **SHA-256**, the functions `wc_InitSha256()`, `wc_Sha256Update()`, and `wc_Sha256Final()` will be used with the structure Sha256.

For **SHA-384**, the functions `wc_InitSha384()`, `wc_Sha384Update()`, and `wc_Sha384Final()` will be used with the structure Sha384.

For **SHA-512**, the functions `wc_InitSha512()`, `wc_Sha512Update()`, and `wc_Sha512Final()` will be used with the structure Sha512.

SHA interleaving (typically only of interest to hardware-acceleration that supports it) is enabled by default, define `NO_WOLFSSL_SHA256_INTERLEAVE` to disable it. Software SHA has always supported interleaving.

10.1.4 BLAKE2b

To use BLAKE2b (a SHA-3 finalist) include the BLAKE2b header `wolfssl/wolfcrypt/blake2.h`. The structure to use is `Blake2b`, which is a typedef. Before using, the hash initialization must be done with the `wc_InitBlake2b()` call. Use `wc_Blake2bUpdate()` to update the hash and `wc_Blake2bFinal()` to retrieve the final hash:

```
byte digest[64];
byte input[64];     /*fill input with data to hash*/

Blake2b b2b;
wc_InitBlake2b(&b2b, 64);

wc_Blake2bUpdate(&b2b, input, sizeof(input));
wc_Blake2bFinal(&b2b, digest, 64);
```

The second parameter to `wc_InitBlake2b()` should be the final digest size. `digest` now contains the digest of the hashed data in buffer.

Example usage can be found in the wolfCrypt test application (`wolfcrypt/test/test.c`), inside the `blake2b_test()` function.

10.1.5 RIPEMD-160

To use RIPEMD-160, include the header `wolfssl/wolfcrypt/ripemd.h`. The structure to use is `RipeMd`, which is a typedef. Before using, the hash initialization must be done with the `wc_InitRipeMd()` call. Use `wc_RipeMdUpdate()` to update the hash and `wc_RipeMdFinal()` to retrieve the final hash

```
byte ripeMdSum[RIPEMD_DIGEST_SIZE];
byte buffer[1024];
/*fill buffer with data to hash*/

RipeMd ripemd;
wc_InitRipeMd(&ripemd);

wc_RipeMdUpdate(&ripemd, buffer, sizeof(buffer));  /*can be called
                                            again and again*/
wc_RipeMdFinal(&ripemd, ripeMdSum);
```

`ripeMdSum` now contains the digest of the hashed data in buffer.

10.2 Keyed Hash Functions

10.2.1 HMAC

wolfCrypt currently provides HMAC for message digest needs. The structure `Hmac` is found in the header `wolfssl/wolfcrypt/hmac.h`. HMAC initialization is done with `wc_HmacSetKey()`. 5 different types are supported with HMAC: MD5, SHA, SHA-256, SHA-384, and SHA-512. Here's an example with SHA-256.

```
Hmac    hmac;
byte    key[24];        /*fill key with keying material*/
byte    buffer[2048];   /*fill buffer with data to digest*/
byte    hmacDigest[SHA256_DIGEST_SIZE];

wc_HmacSetKey(&hmac, SHA256, key, sizeof(key));
wc_HmacUpdate(&hmac, buffer, sizeof(buffer));
wc_HmacFinal(&hmac, hmacDigest);
```

`hmacDigest` now contains the digest of the hashed data in buffer.

10.2.2 GMAC

wolfCrypt also provides GMAC for message digest needs. The structure `Gmac` is found in the header `wolfssl/wolfcrypt/aes.h`, as it is an application AES-GCM. GMAC initialization is done with `wc_GmacSetKey()`.

```
Gmac gmac;
byte    key[16];         /*fill key with keying material*/
byte iv[12];         /*fill iv with an initialization vector*/
byte    buffer[2048];   /*fill buffer with data to digest*/
byte    gmacDigest[16];

wc_GmacSetKey(&gmac, key, sizeof(key));
```

```
wc_GmacUpdate(&gmac, iv, sizeof(iv), buffer, sizeof(buffer),
gmacDigest, sizeof(gmacDigest));
```

gmacDigest now contains the digest of the hashed data in buffer.

10.2.3 Poly1305

wolfCrypt also provides Poly1305 for message digest needs. The structure Poly1305 is found in the header wolfssl/wolfcrypt/poly1305.h. Poly1305 initialization is done with wc_Poly1305SetKey(). The process of setting a key in Poly1305 should be done again, with a new key, when next using Poly1305 after wc_Poly1305Final() has been called.

```
Poly1305    pmac;
byte           key[32];        /*fill key with keying material*/
byte           buffer[2048];   /*fill buffer with data to digest*/
byte           pmacDigest[16];

wc_Poly1305SetKey(&pmac, key, sizeof(key));
wc_Poly1305Update(&pmac, buffer, sizeof(buffer));
wc_Poly1305Final(&pmac, pmacDigest);
```

pmacDigest now contains the digest of the hashed data in buffer.

10.3 Block Ciphers

10.3.1 AES

wolfCrypt provides support for AES with key sizes of 16 bytes (128 bits), 24 bytes (192 bits), or 32 bytes (256 bits). Supported AES modes include CBC, CTR, GCM (GCM-Streaming), OFB, CFB(1, 8, 128), SIV, XTS (XTS-Streaming), GMAC, CMAC, ECB, KW (KeyWrap), and CCM-8.

NOTE: wc_AesInit() should always be called first to initialize the Aes structure, before calling other Aes API functions such as wc_AesSetKey(), and wc_AesGcmSetKey().

CBC mode is supported for both encryption and decryption and is provided through the wc_AesSetKey(), wc_AesCbcEncrypt() and wc_AesCbcDecrypt() functions. Please include the header wolfssl/wolfcrypt/aes.h to use AES. AES has a block size of 16 bytes and the IV should also be 16 bytes. Function usage is usually as follows:

```
Aes enc;
Aes dec;

const byte key[] = {  /*some 24 byte key*/ };
const byte iv[] = { /*some 16 byte iv*/ };

byte plain[32];   /*an increment of 16, fill with data*/
byte cipher[32];

wc_AesInit(&enc, HEAP_HINT, INVALID_DEVID);
wc_AesInit(&dec, HEAP_HINT, INVALID_DEVID);

/*encrypt*/
wc_AesSetKey(&enc, key, sizeof(key), iv, AES_ENCRYPTION);
wc_AesCbcEncrypt(&enc, cipher, plain, sizeof(plain));
```

cipher now contains the ciphertext from the plain text.

```
/*decrypt*/
wc_AesSetKey(&dec, key, sizeof(key), iv, AES_DECRYPTION);
wc_AesCbcDecrypt(&dec, plain, cipher, sizeof(cipher));
```

plain now contains the original plaintext from the ciphertext.

wolfCrypt also supports CTR (Counter), GCM (Galois/Counter), and CCM-8 (Counter with CBC-MAC) modes of operation for AES. When using these modes, like CBC, include the wolfssl/wolfcrypt/aes.h header.

GCM mode is available for both encryption and decryption through the wc_AesGcmSetKey(), wc_AesGcmEncrypt(), and wc_AesGcmDecrypt() functions. For a usage example, see the aesgcm_test() function in <wolfssl_root>/wolfcrypt/test/test.c.

CCM-8 mode is supported for both encryption and decryption through the wc_AesCcmSetKey(), wc_AesCcmEncrypt(), and wc_AesCcmDecrypt() functions. For a usage example, see the aesccm_test() function in <wolfssl_root>/wolfcrypt/test/test.c.

CTR mode is available for both encryption and decryption through the wc_AesCtrEncrypt() function. The encrypt and decrypt actions are identical so the same function is used for both. For a usage example, see the function aes_test() in file wolfcrypt/test/test.c.

10.3.1.1 DES and 3DES wolfCrypt provides support for DES and 3DES (Des3 since 3 is an invalid leading C identifier). To use these include the header wolfssl/wolfcrypt/des.h. The structures you can use are Des and Des3. Initialization is done through wc_Des_SetKey() or wc_Des3_SetKey(). CBC encryption/decryption is provided through wc_Des_CbcEncrypt() / wc_Des_CbcDecrypt() and wc_Des3_CbcEncrypt() / wc_Des3_CbcDecrypt(). Des has a key size of 8 bytes (24 for 3DES) and the block size is 8 bytes, so only pass increments of 8 bytes to encrypt/decrypt functions. If your data isn't in a block size increment you'll need to add padding to make sure it is. Each SetKey() also takes an IV (an initialization vector that is the same size as the key size). Usage is usually like the following:

```
Des3 enc;
Des3 dec;

const byte key[] = {  /*some 24 byte key*/ };
const byte iv[]  = {  /*some 24 byte iv*/  };

byte plain[24];   /*an increment of 8, fill with data*/
byte cipher[24];

/*encrypt*/
wc_Des3_SetKey(&enc, key, iv, DES_ENCRYPTION);
wc_Des3_CbcEncrypt(&enc, cipher, plain, sizeof(plain));
```

cipher now contains the ciphertext from the plain text.

```
/*decrypt*/
wc_Des3_SetKey(&dec, key, iv, DES_DECRYPTION);
wc_Des3_CbcDecrypt(&dec, plain, cipher, sizeof(cipher));
```

plain now contains the original plaintext from the ciphertext.

10.3.1.2 Camellia wolfCrypt provides support for the Camellia block cipher. To use Camellia include the header wolfssl/wolfcrypt/camellia.h. The structure you can use is called Camellia. Initialization is done through wc_CamelliaSetKey(). CBC encryption/decryption is provided through wc_CamelliaCbcEncrypt() and wc_CamelliaCbcDecrypt() while direct encryption/decryption is provided through wc_CamelliaEncryptDirect() and wc_CamelliaDecryptDirect().

For usage examples please see the camellia_test() function in <wolfssl_root>/wolfcrypt/test/test.c.

10.4 Stream Ciphers

10.4.1 ARC4

NOTE: ARC4 is outdated and considered insecure. Please consider using a different stream cipher.

wolfCrypt supports ARC4 through the header wolfssl/wolfcrypt/arc4.h. Usage is simpler than block ciphers because there is no block size and the key length can be any length. The following is a typical usage of ARC4.

```
Arc4 enc;
Arc4 dec;

const byte key[] = {  /*some key any length*/};

byte plain[27];   /*no size restriction, fill with data*/
byte cipher[27];

/*encrypt*/
wc_Arc4SetKey(&enc, key, sizeof(key));
wc_Arc4Process(&enc, cipher, plain, sizeof(plain));
```

cipher now contains the ciphertext from the plain text.

```
/*decrypt*/
wc_Arc4SetKey(&dec, key, sizeof(key));
wc_Arc4Process(&dec, plain, cipher, sizeof(cipher));
```

plain now contains the original plaintext from the ciphertext.

10.4.2 ChaCha

ChaCha with 20 rounds is slightly faster than ARC4 while maintaining a high level of security. To use it with wolfCrypt, please include the header wolfssl/wolfcrypt/chacha.h. ChaCha typically uses 32 byte keys (256 bit) but can also use 16 byte keys (128 bits).

```
ChaCha enc;
ChaCha dec;

const byte key[] = {  /*some key 32 bytes*/};
const byte iv[]  = {  /*some iv 12 bytes*/ };

byte plain[37];   /*no size restriction, fill with data*/
byte cipher[37];

/*encrypt*/
wc_Chacha_SetKey(&enc, key, keySz);
wc_Chacha_SetIV(&enc, iv, counter); /*counter is the start block
                        counter is usually set as 0*/
wc_Chacha_Process(&enc, cipher, plain, sizeof(plain));
```

cipher now contains the ciphertext from the plain text.

```
/*decrypt*/
wc_Chacha_SetKey(&enc, key, keySz);
```

```
wc_Chacha_SetIV(&enc, iv, counter);
wc_Chacha_Process(&enc, plain, cipher, sizeof(cipher));
```

plain now contains the original plaintext from the ciphertext.

wc_Chacha_SetKey only needs to be set once but for each packet of information sent wc_Chacha_SetIV() must be called with a new iv (nonce). Counter is set as an argument to allow for partially decrypting/encrypting information by starting at a different block when performing the encrypt/decrypt process, but in most cases is set to 0. **ChaCha should not be used without a mac algorithm (e.g. Poly1305 , hmac).**

10.5 Public Key Cryptography

10.5.1 RSA

wolfCrypt provides support for RSA through the header wolfssl/wolfcrypt/rsa.h. There are two types of RSA keys, public and private. A public key allows anyone to encrypt something that only the holder of the private key can decrypt. It also allows the private key holder to sign something and anyone with a public key can verify that only the private key holder actually signed it. Usage is usually like the following:

```
RsaKey rsaPublicKey;

byte publicKeyBuffer[]  = { /*holds the raw data from the key, maybe
                          from a file like RsaPublicKey.der*/ };
word32 idx = 0;            /*where to start reading into the buffer*/

wc_RsaPublicKeyDecode(publicKeyBuffer, &idx, &rsaPublicKey,
 ↪  sizeof(publicKeyBuffer));

byte in[] = { /*plain text to encrypt*/ };
byte out[128];
RNG rng;

wc_InitRng(&rng);

word32 outLen = wc_RsaPublicEncrypt(in, sizeof(in), out, sizeof(out),
 ↪  &rsaPublicKey, &rng);
```

Now out holds the ciphertext from the plain text in. wc_RsaPublicEncrypt() will return the length in bytes written to out or a negative number in case of an error. wc_RsaPublicEncrypt() needs a RNG (Random Number Generator) for the padding used by the encryptor and it must be initialized before it can be used. To make sure that the output buffer is large enough to pass you can first call wc_RsaEncryptSize() which will return the number of bytes that a successful call to wc_RsaPublicEncrypt() will write.

In the event of an error, a negative return from wc_RsaPublicEncrypt(), or wc_RsaPublicKeyDecode() for that matter, you can call wc_ErrorString() to get a string describing the error that occurred.

```
void wc_ErrorString(int error, char* buffer);
```

Make sure that buffer is at least WOLFSSL_MAX_ERROR_SZ bytes (80).

Now to decrypt out:

```
RsaKey rsaPrivateKey;

byte privateKeyBuffer[] = { /*hold the raw data from the key, maybe
```

```
                                from a file like RsaPrivateKey.der*/ };
word32 idx = 0;             /*where to start reading into the buffer*/

wc_RsaPrivateKeyDecode(privateKeyBuffer, &idx, &rsaPrivateKey,
                       sizeof(privateKeyBuffer));

byte plain[128];
word32 plainSz = wc_RsaPrivateDecrypt(out, outLen, plain,
                       sizeof(plain), &rsaPrivateKey);
```

Now plain will hold plainSz bytes or an error code. For complete examples of each type in wolfCrypt please see the file `wolfcrypt/test/test.c`. Note that the `wc_RsaPrivateKeyDecode` function only accepts keys in raw DER format.

10.5.2 DH (Diffie-Hellman)

wolfCrypt provides support for Diffie-Hellman through the header `wolfssl/wolfcrypt/dh.h`. The Diffie-Hellman key exchange algorithm allows two parties to establish a shared secret key. Usage is usually similar to the following example, where **sideA** and **sideB** designate the two parties.

In the following example, dhPublicKey contains the Diffie-Hellman public parameters signed by a Certificate Authority (or self-signed). `privA` holds the generated private key for sideA, pubA holds the generated public key for sideA, and `agreeA` holds the mutual key that both sides have agreed on.

```
DhKey   dhPublicKey;
word32 idx = 0;  /*where to start reading into the
                   publicKeyBuffer*/
word32 pubASz, pubBSz, agreeASz;
byte   tmp[1024];
RNG    rng;

byte privA[128];
byte pubA[128];
byte agreeA[128];

wc_InitDhKey(&dhPublicKey);

byte publicKeyBuffer[] = { /*holds the raw data from the public key
                            parameters, maybe from a file like
                            dh1024.der*/ };
wc_DhKeyDecode(publicKeyBuffer, &idx, &dhPublicKey, sizeof(publicKeyBuffer));
wc_InitRng(&rng);  /*Initialize random number generator*/
```

`wc_DhGenerateKeyPair()` will generate a public and private DH key based on the initial public parameters in dhPublicKey.

```
wc_DhGenerateKeyPair(&dhPublicKey, &rng, privA, &privASz,
            pubA, &pubASz);
```

After sideB sends their public key (pubB) to sideA, sideA can then generate the mutually-agreed key(agreeA) using the `wc_DhAgree()` function.

```
wc_DhAgree(&dhPublicKey, agreeA, &agreeASz, privA, privASz,
           pubB, pubBSz);
```

Now, agreeA holds sideA's mutually-generated key (of size `agreeASz` bytes). The same process will have been done on sideB.

For a complete example of Diffie-Hellman in wolfCrypt, see the file `wolfcrypt/test/test.c`.

10.5.3 EDH (Ephemeral Diffie-Hellman)

A wolfSSL server can do Ephemeral Diffie-Hellman. No build changes are needed to add this feature, though an application will have to register the ephemeral group parameters on the server side to enable the EDH cipher suites. A new API can be used to do this:

```
int wolfSSL_SetTmpDH(WOLFSSL* ssl, unsigned char* p,
                     int pSz,unsigned char* g,int gSz);
```

The example server and echoserver use this function from `SetDH()`.

10.5.4 DSA (Digital Signature Algorithm)

wolfCrypt provides support for DSA and DSS through the header `wolfssl/wolfcrypt/dsa.h`. DSA allows for the creation of a digital signature based on a given data hash. DSA uses the SHA hash algorithm to generate a hash of a block of data, then signs that hash using the signer's private key. Standard usage is similar to the following.

We first declare our DSA key structure (`key`), initialize our initial message (`message`) to be signed, and initialize our DSA key buffer (`dsaKeyBuffer`).

```
DsaKey key;
Byte   message[]      = { /*message data to sign*/ }
byte   dsaKeyBuffer[] = { /*holds the raw data from the DSA key,
                            maybe from a file like dsa512.der*/ }
```

We then declare our SHA structure (`sha`), random number generator (`rng`), array to store our SHA hash (`hash`), array to store our signature (`signature`), `idx` (to mark where to start reading in our `dsaKeyBuffer`), and an int (`answer`) to hold our return value after verification.

```
Sha    sha;
RNG    rng;
byte   hash[SHA_DIGEST_SIZE];
byte   signature[40];
word32 idx = 0;
int    answer;
```

Set up and create the SHA hash. For more information on wolfCrypt's SHA algorithm, see SHA / SHA-224 / SHA-256 / SHA-384 / SHA-512. The SHA hash of `message` is stored in the variable `hash`.

```
wc_InitSha(&sha);
wc_ShaUpdate(&sha, message, sizeof(message));
wc_ShaFinal(&sha, hash);
```

Initialize the DSA key structure, populate the structure key value, and initialize the random number generator (`rng`).

```
wc_InitDsaKey(&key);
wc_DsaPrivateKeyDecode(dsaKeyBuffer, &idx, &key,
                       sizeof(dsaKeyBuffer));
wc_InitRng(&rng);
```

The `wc_DsaSign()` function creates a signature (`signature`) using the DSA private key, hash value, and random number generator.

```
wc_DsaSign(hash, signature, &key, &rng);
```

To verify the signature, use `wc_DsaVerify()`. If verification is successful, answer will be equal to "**1**". Once finished, free the DSA key structure using `wc_FreeDsaKey()`.

```
wc_DsaVerify(hash, signature, &key, &answer);
wc_FreeDsaKey(&key);
```

11 SSL Tutorial

11.1 Introduction

The wolfSSL embedded SSL/TLS library can easily be integrated into your existing application or device to provide enhanced communication security through the addition of SSL and TLS. wolfSSL has been targeted at embedded and RTOS environments, and as such, offers a minimal footprint while maintaining excellent performance. Minimum build sizes for wolfSSL range between 20-100kB depending on the selected build options and platform being used.

The goal of this tutorial is to walk through the integration of SSL and TLS into a simple application. Hopefully the process of going through this tutorial will also lead to a better understanding of SSL in general. This tutorial uses wolfSSL in conjunction with simple echoserver and echoclient examples to keep things as simple as possible while still demonstrating the general procedure of adding SSL support to an application. The echoserver and echoclient examples have been taken from the popular book titled Unix Network Programming, Volume 1, 3rd Edition by Richard Stevens, Bill Fenner, and Andrew Rudoff.

This tutorial assumes that the reader is comfortable with editing and compiling C code using the GNU GCC compiler as well as familiar with the concepts of public key encryption. Please note that access to the Unix Network Programming book is not required for this tutorial.

11.1.1 Examples Used in this Tutorial

- echoclient - Figure 5.4, Page 124
- echoserver - Figure 5.12, Page 139

11.2 Quick Summary of SSL/TLS

TLS (Transport Layer Security) and **SSL** (Secure Sockets Layer) are cryptographic protocols that allow for secure communication across a number of different transport protocols. The primary transport protocol used is TCP/IP. The most recent version of SSL/TLS is TLS 1.3. wolfSSL supports SSL 3.0, TLS 1.0, 1.1, 1.2, 1.3 in addition to DTLS 1.0, 1.2, and 1.3

SSL and TLS sit between the Transport and Application layers of the OSI model, where any number of protocols (including TCP/IP, Bluetooth, etc.) may act as the underlying transport medium. Application protocols are layered on top of SSL and can include protocols such as HTTP, FTP, and SMTP. A diagram of how SSL fits into the OSI model, as well as a simple diagram of the SSL handshake process can be found in Appendix A.

11.3 Getting the Source Code

All of the source code used in this tutorial can be downloaded from the wolfSSL website, specifically from the following location. The download contains both the original and completed source code for both the echoserver and echoclient used in this tutorial. Specific contents are listed below the link.

https://www.wolfssl.com/documentation/ssl-tutorial-2.5.zip

The downloaded ZIP file has the following structure:

```
/finished_src
    /certs  (Certificate files)
    /echoclient (Completed echoclient code)
    /echoserver (Completed echoserver code)
    /include    (Modified unp.h)
    /lib    (Library functions)
```

```
/original_src
    /echoclient (Starting echoclient code)
    /echoserver (Starting echoserver code)
    /include    (Modified unp.h)
    /lib    (Library functions)
README
```

11.4 Base Example Modifications

This tutorial, and the source code that accompanies it, have been designed to be as portable as possible across platforms. Because of this, and because we want to focus on how to add SSL and TLS into an application, the base examples have been kept as simple as possible. Several modifications have been made to the examples taken from Unix Network Programming in order to either remove unnecessary complexity or increase the range of platforms supported. If you believe there is something we could do to increase the portability of this tutorial, please let us know at support@wolfssl.com.

The following is a list of modifications that were made to the original echoserver and echoclient examples found in the above listed book.

11.4.1 Modifications to the echoserver (tcpserv04.c)

- Removed call to the `fork()` function because `fork()` is not supported by Windows. The result of this is an echoserver which only accepts one client simultaneously. Along with this removal, Signal handling was removed.
- Moved `str_echo()` function from `str_echo.c` file into `tcpserv04.c` file
- Added a printf statement to view the client address and the port we have connected through:

```
printf("Connection from %s, port %d\n",
        inet_ntop(AF_INET, &cliaddr.sin_addr, buff, sizeof(buff)),
        ntohs(cliaddr.sin_port));
```

- Added a call to `setsockopt()` after creating the listening socket to eliminate the “Address already in use” bind error.
- Minor adjustments to clean up newer compiler warnings

11.4.2 Modifications to the echoclient (tcpcli01.c)

- Moved `str_cli()` function from `str_cli.c` file into `tcpcli01.c` file
- Minor adjustments to clean up newer compiler warnings

11.4.3 Modifications to unp.h header

- This header was simplified to contain only what is needed for this example.

Please note that in these source code examples, certain functions will be capitalized. For example, `Fputs()` and `Writen()`. The authors of Unix Network Programming have written custom wrapper functions for normal functions in order to cleanly handle error checking. For a more thorough explanation of this, please see **Section 1.4** (page 11) in the *Unix Network Programming* book.

11.5 Building and Installing wolfSSL

Before we begin, download the example code (`echoserver` and `echoclient`) from the Getting the Source Code section, above. This section will explain how to download, configure, and install the wolfSSL embedded SSL/TLS library on your system.

You will need to download and install the most recent version of wolfSSL from the wolfSSL download page.

For a full list of available build options, see the Building wolfSSL guide. wolfSSL was written with portability in mind, and should generally be easy to build on most systems. If you have difficulty building wolfSSL, please feel free to ask for support on the wolfSSL product support forums.

When building wolfSSL on Linux, *BSD, OS X, Solaris, or other* nix like systems, you can use the autoconf system. For Windows-specific instructions, please refer to the Building wolfSSL section of the wolfSSL Manual. To configure and build wolfSSL, run the following two commands from the terminal. Any desired build options may be appended to `./configure` (ex: `./configure --enable-opensslextra`):

```
./configure
make
```

To install wolfSSL, run:

```
sudo make install
```

This will install wolfSSL headers into `/usr/local/include/wolfssl` and the wolfSSL libraries into `/usr/local/lib` on your system. To test the build, run the testsuite application from the wolfSSL root directory:

```
./testsuite/testsuite.test
```

A set of tests will be run on wolfCrypt and wolfSSL to verify it has been installed correctly. After a successful run of the testsuite application, you should see output similar to the following:

```
------------------------------------------------------------------------------
 wolfSSL version 5.7.0
------------------------------------------------------------------------------
error    test passed!
MEMORY   test passed!
base64   test passed!
asn      test passed!
RANDOM   test passed!
MD5      test passed!
SHA      test passed!
SHA-224  test passed!
SHA-256  test passed!
SHA-384  test passed!
SHA-512  test passed!
SHA-512/224  test passed!
SHA-512/256  test passed!
SHA-3    test passed!
Hash     test passed!
HMAC-MD5 test passed!
HMAC-SHA test passed!
HMAC-SHA224 test passed!
HMAC-SHA256 test passed!
HMAC-SHA384 test passed!
HMAC-SHA512 test passed!
```

```
HMAC-SHA3   test passed!
HMAC-KDF    test passed!
PRF         test passed!
TLSv1.3 KDF test passed!
GMAC     test passed!
Chacha   test passed!
POLY1305 test passed!
ChaCha20-Poly1305 AEAD test passed!
AES      test passed!
AES192   test passed!
AES256   test passed!
AES-GCM  test passed!
RSA      test passed!
DH       test passed!
PWDBASED test passed!
ECC      test passed!
logging  test passed!
time     test passed!
mutex    test passed!
memcb    test passed!
Test complete

Running simple test
SSL version is TLSv1.2
SSL version is TLSv1.2
SSL cipher suite is TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
SSL curve name is SECP256R1
SSL cipher suite is TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
SSL curve name is SECP256R1
Client message: hello wolfssl!
I hear you fa shizzle!

Running TLS test
sending server shutdown command: quit!
client sent quit command: shutting down!
ciphers = TLS13-AES128-GCM-SHA256:TLS13-AES256-GCM-SHA384:TLS13-CHACHA20-
   POLY1305-SHA256:DHE-RSA-AES128-SHA:DHE-RSA-AES256-SHA:ECDHE-RSA-AES128-SHA:
   ECDHE-RSA-AES256-SHA:ECDHE-ECDSA-AES128-SHA:ECDHE-ECDSA-AES256-SHA:DHE-RSA-
   AES128-SHA256:DHE-RSA-AES256-SHA256:DHE-RSA-AES128-GCM-SHA256:DHE-RSA-
   AES256-GCM-SHA384:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-RSA-AES256-GCM-SHA384:
   ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-
   AES128-SHA256:ECDHE-ECDSA-AES128-SHA256:ECDHE-RSA-AES256-SHA384:ECDHE-ECDSA
   -AES256-SHA384:ECDHE-RSA-CHACHA20-POLY1305:ECDHE-ECDSA-CHACHA20-POLY1305:
   DHE-RSA-CHACHA20-POLY1305:ECDHE-RSA-CHACHA20-POLY1305-OLD:ECDHE-ECDSA-
   CHACHA20-POLY1305-OLD:DHE-RSA-CHACHA20-POLY1305-OLD
33bc1a4570f4f1abccd5c48aace529b01a42ab51293954a297796e90d20970f0  input
33bc1a4570f4f1abccd5c48aace529b01a42ab51293954a297796e90d20970f0  /var/folders
   /dy/888x7r7d6dgcqw4840l32tpw0000gp/T//testsuite-output-9Ymbuv

All tests passed!
```

Now that wolfSSL has been installed, we can begin modifying the example code to add SSL functionality. We will first begin by adding SSL to the echoclient and subsequently move on to the echoserver.

11.6 Initial Compilation

To compile and run the example echoclient and echoserver code from the SSL Tutorial source bundle, you can use the included Makefiles. Change directory (cd) to either the echoclient or echoserver directory and run:

```
make
```

This will compile the example code and produce an executable named either echoserver or echoclient depending on which one is being built. The GCC command which is used in the Makefile can be seen below. If you want to build one of the examples without using the supplied Makefile, change directory to the example directory and replace `tcpcli01.c` (echoclient) or `tcpserv04.c` (echoserver) in the following command with correct source file for the example:

```
gcc -o echoserver ../lib/*.c tcpserv04.c -I ../include
```

This will compile the current example into an executable, creating either an "echoserver" or "echoclient" application. To run one of the examples after it has been compiled, change your current directory to the desired example directory and start the application. For example, to start the echoserver use:

```
./echoserver
```

You may open a second terminal window to test the echoclient on your local host and you will need to supply the IP address of the server when starting the application, which in our case will be 127.0.0.1. Change your current directory to the "echoclient" directory and run the following command. Note that the echoserver must already be running:

```
./echoclient 127.0.0.1
```

Once you have both the echoserver and echoclient running, the echoserver should echo back any input that it receives from the echoclient. To exit either the echoserver or echoclient, use *Ctrl + C* to quit the application. Currently, the data being echoed back and forth between these two examples is being sent in the clear - easily allowing anyone with a little bit of skill to inject themselves in between the client and server and listen to your communication.

11.7 Libraries

The wolfSSL library, once compiled, is named `libwolfssl`, and unless otherwise configured the wolfSSL build and install process creates only a shared library under the following directory. Both shared and static libraries may be enabled or disabled by using the appropriate build options:

```
/usr/local/lib
```

The first step we need to do is link the wolfSSL library to our example applications. Modifying the GCC command (using the echoserver as an example), gives us the following new command. Since wolfSSL installs header files and libraries in standard locations, the compiler should be able to find them without explicit instructions (using `-l` or `-L`). Note that by using `-lwolfssl` the compiler will automatically choose the correct type of library (static or shared):

```
gcc -o echoserver ../lib/*.c tcpserv04.c -I ../include -lm -lwolfssl
```

11.8 Headers

The first thing we will need to do is include the wolfSSL native API header in both the client and the server. In the `tcpcli01.c` file for the client and the `tcpserv04.c` file for the server add the following line near the top:

```
#include <wolfssl/ssl.h>
```

11.9 Startup/Shutdown

Before we can use wolfSSL in our code, we need to initialize the library and the WOLFSSL_CTX. wolfSSL is initialized by calling wolfSSL_Init(). This must be done first before anything else can be done with the library.

11.10 WOLFSSL_CTX Factory

The WOLFSSL_CTX structure (wolfSSL Context) contains global values for each SSL connection, including certificate information. A single WOLFSSL_CTX can be used with any number of WOLFSSL objects created. This allows us to load certain information, such as a list of trusted CA certificates only once.

To create a new WOLFSSL_CTX, use wolfSSL_CTX_new(). This function requires an argument which defines the SSL or TLS protocol for the client or server to use. There are several options for selecting the desired protocol. wolfSSL currently supports SSL 3.0, TLS 1.0, TLS 1.1, TLS 1.2, TLS 1.3, DTLS 1.0, DTLS 1.2, and DTLS 1.3. Each of these protocols have a corresponding function that can be used as an argument to wolfSSL_CTX_new(). The possible client and server protocol options are shown below. SSL 2.0 is not supported by wolfSSL because it has been insecure for several years.

EchoClient:

- wolfSSLv3_client_method(); - SSL 3.0
- wolfTLSv1_client_method(); - TLS 1.0
- wolfTLSv1_1_client_method(); - TLS 1.1
- wolfTLSv1_2_client_method(); - TLS 1.2
- wolfTLSv1_3_client_method(); - TLS 1.3
- wolfSSLv23_client_method(); - Use highest version possible from SSL 3.0 - TLS 1.3
- wolfDTLSv1_client_method(); - DTLS 1.0
- wolfDTLSv1_2_client_method_ex(); - DTLS 1.2
- wolfDTLSv1_3_client_method_ex(); - DTLS 1.3

EchoServer:

- wolfSSLv3_server_method(); - SSL 3.0
- wolfTLSv1_server_method(); - TLS 1.0
- wolfTLSv1_1_server_method(); - TLS 1.1
- wolfTLSv1_2_server_method(); - TLS 1.2
- wolfTLSv1_3_server_method(); - TLS 1.3
- wolfSSLv23_server_method(); - Allow clients to connect with TLS 1.0+
- wolfDTLSv1_server_method(); - DTLS 1.0
- wolfDTLSv1_2_server_method(); - DTLS 1.2
- wolfDTLSv1_3_server_method(); - DTLS 1.3

We need to load our CA (Certificate Authority) certificate into the WOLFSSL_CTX so that the when the echoclient connects to the echoserver, it is able to verify the server's identity. To load the CA certificates into the WOLFSSL_CTX, use wolfSSL_CTX_load_verify_locations(). This function requires three arguments: a WOLFSSL_CTX pointer, a certificate file, and a path value. The path value points to a directory which should contain CA certificates in PEM format. When looking up certificates, wolfSSL will look at the certificate file value before looking in the path location. In this case, we don't need to specify a certificate path because we will specify one CA file - as such we use NULL for the path argument. The wolfSSL_CTX_load_verify_locations function returns either SSL_SUCCESS or SSL_FAILURE:

```
wolfSSL_CTX_load_verify_locations(WOLFSSL_CTX* ctx, const char* file, const
↪  char* path)
```

Putting these things together (library initialization, protocol selection, and CA certificate), we have the following. Here, we choose to use TLS 1.2:

EchoClient:

```
    WOLFSSL_CTX* ctx;

    wolfSSL_Init();/* Initialize wolfSSL */

    /* Create the WOLFSSL_CTX */
    if ( (ctx = wolfSSL_CTX_new(wolfTLSv1_2_client_method())) == NULL) {
        fprintf(stderr, "wolfSSL_CTX_new error.\n");
        exit(EXIT_FAILURE);
    }

    /* Load CA certificates into WOLFSSL_CTX */
    if (wolfSSL_CTX_load_verify_locations(ctx, "../certs/ca-cert.pem", NULL) !=
        SSL_SUCCESS) {
        fprintf(stderr, "Error loading ../certs/ca-cert.pem, "
            "please check the file.\n");
        exit(EXIT_FAILURE);
    }
```

Add the above code to `tcpcli01.c` in `main()` after the variable definitions and the check that the user has started the client with an IP address.

EchoServer:

When loading certificates into the `WOLFSSL_CTX`, the server certificate and key file should be loaded in addition to the CA certificate. This will allow the server to send the client its certificate for identification verification:

```
WOLFSSL_CTX* ctx;

wolfSSL_Init();  /* Initialize wolfSSL */

/* Create the WOLFSSL_CTX */
if ( (ctx = wolfSSL_CTX_new(wolfTLSv1_2_server_method())) == NULL) {
    fprintf(stderr, "wolfSSL_CTX_new error.\n");
    exit(EXIT_FAILURE);
}

/* Load CA certificates into WOLFSSL_CTX */
if (wolfSSL_CTX_load_verify_locations(ctx, "../certs/ca-cert.pem", NULL) !=
         SSL_SUCCESS) {
    fprintf(stderr, "Error loading ../certs/ca-cert.pem, "
        "please check the file.\n");
    exit(EXIT_FAILURE);
}

/* Load server certificates into WOLFSSL_CTX */
if (wolfSSL_CTX_use_certificate_file(ctx, "../certs/server-cert.pem",
        SSL_FILETYPE_PEM) != SSL_SUCCESS) {
    fprintf(stderr, "Error loading ../certs/server-cert.pem, please"
        "check the file.\n");
    exit(EXIT_FAILURE);
}

/* Load keys */
if (wolfSSL_CTX_use_PrivateKey_file(ctx, "../certs/server-key.pem",
```

```
        SSL_FILETYPE_PEM) != SSL_SUCCESS) {
    fprintf(stderr, "Error loading ../certs/server-key.pem, please check"
        "the file.\n");
    exit(EXIT_FAILURE);
}
```

The code shown above should be added to the beginning of `tcpserv04.c` after the variable definitions in `main()`. A version of the finished code is included in the SSL tutorial ZIP file for reference.

Now that wolfSSL and the `WOLFSSL_CTX` have been initialized, make sure that the `WOLFSSL_CTX` object and the wolfSSL library are freed when the application is completely done using SSL/TLS. In both the client and the server, the following two lines should be placed at the end of the `main()` function (in the client right before the call to `exit()`):

```
wolfSSL_CTX_free(ctx);
wolfSSL_Cleanup();
```

11.11 WOLFSSL Object

11.11.1 EchoClient

A WOLFSSL object needs to be created after each TCP connect and the socket file descriptor needs to be associated with the session. In the echoclient example, we will do this after the call to `connect()`, shown below:

```
/* Connect to socket file descriptor */
connect(sockfd, (SA *) &servaddr, sizeof(servaddr));
```

Directly after connecting, create a new `WOLFSSL` object using the `wolfSSL_new()` function. This function returns a pointer to the `WOLFSSL` object if successful or `NULL` in the case of failure. We can then associate the socket file descriptor (`sockfd`) with the new `WOLFSSL` object (`ssl`):

```
/* Create WOLFSSL object */
WOLFSSL* ssl;

if( (ssl = wolfSSL_new(ctx)) == NULL) {
    fprintf(stderr, "wolfSSL_new error.\n");
    exit(EXIT_FAILURE);
}

wolfSSL_set_fd(ssl, sockfd);
```

One thing to notice here is that we haven't made a call to the `wolfSSL_connect()` function. `wolfSSL_connect()` initiates the SSL/TLS handshake with the server, and is called during `wolfSSL_read()` if it hadn't been called previously. In our case, we don't explicitly call `wolfSSL_connect()`, as we let our first `wolfSSL_read()` do it for us.

11.11.2 EchoServer

At the end of the for loop in the main method, insert the WOLFSSL object and associate the socket file descriptor (`connfd`) with the `WOLFSSL` object (`ssl`), just as with the client:

```
/* Create WOLFSSL object */
WOLFSSL* ssl;

if ( (ssl = wolfSSL_new(ctx)) == NULL) {
            fprintf(stderr, "wolfSSL_new error.\n");
            exit(EXIT_FAILURE);
```

```
}

wolfSSL_set_fd(ssl, connfd);
```

Again, a WOLFSSL object needs to be created after each TCP Connect and the socket file descriptor needs to be associated with the session.

11.12 Sending/Receiving Data

11.12.1 Sending with EchoClient

The next step is to begin sending data securely. Take note that in the echoclient example, the `main()` function hands off the sending and receiving work to `str_cli()`. The `str_cli()` function is where our function replacements will be made. First we need access to our `WOLFSSL` object in the `str_cli()` function, so we add another argument and pass the ssl variable to `str_cli()`. Because the `WOLFSSL` object is now going to be used inside of the `str_cli()` function, we remove the `sockfd` parameter. The new `str_cli()` function signature after this modification is shown below:

```
void str_cli(FILE *fp, WOLFSSL* ssl)
```

In the `main()` function, the new argument (`ssl`) is passed to `str_cli()`:

```
str_cli(stdin, ssl);
```

Inside the `str_cli()` function, `Writen()` and `Readline()` are replaced with calls to `wolfSSL_write()` and `wolfSSL_read()` functions, and the `WOLFSSL` object (`ssl`) is used instead of the original file descriptor(`sockfd`). The new `str_cli()` function is shown below. Notice that we now need to check if our calls to `wolfSSL_write` and `wolfSSL_read` were successful.

The authors of the Unix Programming book wrote error checking into their `Writen()` function which we must make up for after it has been replaced. We add a new int variable, `n`, to monitor the return value of `wolfSSL_read` and before printing out the contents of the buffer, `recvline`, the end of our read data is marked with a `\0`:

```
void
str_cli(FILE *fp, WOLFSSL* ssl)
{
    char  sendline[MAXLINE], recvline[MAXLINE];
    int   n = 0;

    while (Fgets(sendline, MAXLINE, fp) != NULL) {

        if(wolfSSL_write(ssl, sendline, strlen(sendline)) !=
                    strlen(sendline)){
            err_sys("wolfSSL_write failed");
        }

        if ((n = wolfSSL_read(ssl, recvline, MAXLINE)) <= 0)
            err_quit("wolfSSL_read error");

        recvline[n] = '\0';
        Fputs(recvline, stdout);
    }
}
```

The last thing to do is free the `WOLFSSL` object when we are completely done with it. In the `main()` function, right before the line to free the `WOLFSSL_CTX`, call to `wolfSSL_free()`:

```
str_cli(stdin, ssl);

wolfSSL_free(ssl);      /* Free WOLFSSL object */
wolfSSL_CTX_free(ctx);  /* Free WOLFSSL_CTX object */
wolfSSL_Cleanup();      /* Free wolfSSL */
```

11.12.2 Receiving with EchoServer

The echo server makes a call to `str_echo()` to handle reading and writing (whereas the client made a call to `str_cli()`). As with the client, modify `str_echo()` by replacing the sockfd parameter with a `WOLFSSL` object (`ssl`) parameter in the function signature:

```
void str_echo(WOLFSSL* ssl)
```

Replace the calls to `Read()` and `Writen()` with calls to the `wolfSSL_read()` and `wolfSSL_write()` functions. The modified `str_echo()` function, including error checking of return values, is shown below. Note that the type of the variable n has been changed from `ssize_t` to `int` in order to accommodate for the change from `read()` to `wolfSSL_read()`:

```
void
str_echo(WOLFSSL* ssl)
{
    int n;
    char buf[MAXLINE];

    while ( (n = wolfSSL_read(ssl, buf, MAXLINE)) > 0) {
        if(wolfSSL_write(ssl, buf, n) != n) {
            err_sys("wolfSSL_write failed");
        }
    }

    if( n < 0 )
        printf("wolfSSL_read error = %d\n", wolfSSL_get_error(ssl,n));
    else if( n == 0 )
        printf("The peer has closed the connection.\n");
}
```

In `main()` call the `str_echo()` function at the end of the for loop (soon to be changed to a while loop). After this function, inside the loop, make calls to free the `WOLFSSL` object and close the connfd socket:

```
str_echo(ssl);                  /* process the request */

wolfSSL_free(ssl);              /* Free WOLFSSL object */
Close(connfd);
```

We will free the `ctx` and cleanup before the call to exit.

11.13 Signal Handling

11.13.1 Echoclient / Echoserver

In the echoclient and echoserver, we will need to add a signal handler for when the user closes the app by using “Ctrl+C”. The echo server is continually running in a loop. Because of this, we need to provide a way to break that loop when the user presses “Ctrl+C”. To do this, the first thing we need to do is change our loop to a while loop which terminates when an exit variable (cleanup) is set to true.

First, define a new static int variable called `cleanup` at the top of `tcpserv04.c` right after the `#include` statements:

```
static int cleanup;  /* To handle shutdown */
```

Modify the echoserver loop by changing it from a for loop to a while loop:

```
while(cleanup != 1)
{
    /* echo server code here */
}
```

For the echoserver we need to disable the operating system from restarting calls which were being executed before the signal was handled after our handler has finished. By disabling these, the operating system will not restart calls to `accept()` after the signal has been handled. If we didn't do this, we would have to wait for another client to connect and disconnect before the echoserver would clean up resources and exit. To define the signal handler and turn off SA_RESTART, first define `act` and `oact` structures in the echoserver's `main()` function:

```
struct sigaction     act, oact;
```

Insert the following code after variable declarations, before the call to `wolfSSL_Init()` in the main function:

```
/* Signal handling code */
struct sigaction act, oact;          /* Declare the sigaction structs    */
act.sa_handler = sig_handler;        /* Tell act to use sig_handler      */
sigemptyset(&act.sa_mask);           /* Tells act to exclude all sa_mask *
                                      * signals during execution of      *
                                      * sig_handler.                     */
act.sa_flags = 0;                    /* States that act has a special    *
                                      * flag of 0                        */
sigaction(SIGINT, &act, &oact);      /* Tells the program to use (o)act  *
                                      * on a signal or interrupt         */
```

The echoserver's sig_handler function is shown below:

```
void sig_handler(const int sig)
{
    printf("\nSIGINT handled.\n");
    cleanup = 1;
    return;
}
```

That's it - the echoclient and echoserver are now enabled with TLSv1.2!!

What we did:

- Included the wolfSSL headers
- Initialized wolfSSL
- Created a `WOLFSSL_CTX` structure in which we chose what protocol we wanted to use
- Created a `WOLFSSL` object to use for sending and receiving data
- Replaced calls to `Writen()` and `Readline()` with `wolfSSL_write()` and `wolfSSL_read()`
- Freed `WOLFSSL`, `WOLFSSL_CTX`
- Made sure we handled client and server shutdown with signal handler

There are many more aspects and methods to configure and control the behavior of your SSL connections. For more detailed information, please see additional wolfSSL documentation and resources.

Once again, the completed source code can be found in the downloaded ZIP file at the top of this section.

COPYRIGHT ©2024 wolfSSL Inc.

11.14 Certificates

For testing purposes, you may use the certificates provided by wolfSSL. These can be found in the wolfSSL download, and specifically for this tutorial, they can be found in the `finished_src` folder.

For production applications, you should obtain correct and legitimate certificates from a trusted certificate authority.

11.15 Conclusion

This tutorial walked through the process of integrating the wolfSSL embedded SSL/TLS library into a simple client and server application. Although this example is simple, the same principles may be applied for adding SSL or TLS into your own application. The wolfSSL embedded SSL/TLS library provides all the features you would need in a compact and efficient package that has been optimized for both size and speed.

Being dual licensed under GPLv3 and standard commercial licensing, you are free to download the wolfSSL source code directly from our website. Feel free to post to our support forums (https://www.wolfssl.com/forums) with any questions or comments you might have. If you would like more information about our products, please contact facts@wolfssl.com.

We welcome any feedback you have on this SSL tutorial. If you believe it could be improved or enhanced in order to make it either more useful, easier to understand, or more portable, please let us know at support@wolfssl.com.

COPYRIGHT ©2024 wolfSSL Inc.

12 Best Practices for Embedded Devices

12.1 Creating Private Keys

Embedding a private key into firmware allows anyone to extract the key and turns an otherwise secure connection into something nothing more secure than TCP.

We have a few ideas about creating private keys for SSL enabled devices.

1. Each device acting as a server should have a unique private key, just like in the non-embedded world.
2. If the key can't be placed onto the device before delivery, have it generated during setup.
3. If the device lacks the power to generate its own key during setup, have the client setting up the device generate the key and send it to the device.
4. If the client lacks the ability to generate a private key, have the client retrieve a unique private key over an SSL/TLS connection from the devices known website (for example).

wolfSSL (formerly CyaSSL) can be used in all of these steps to help ensure an embedded device has a secure unique private key. Taking these steps will go a long way towards securing the SSL connection itself.

12.2 Digitally Signing and Authenticating with wolfSSL

wolfSSL is a popular tool for digitally signing applications, libraries, or files prior to loading them on embedded devices. Most desktop and server operating systems allow creation of this type of functionality through system libraries, but stripped down embedded operating systems do not. The reason that embedded RTOS environments do not include digital signature functionality is because it has historically not been a requirement for most embedded applications. In today's world of connected devices and heightened security concerns, digitally signing what is loaded onto your embedded or mobile device has become a top priority.

Examples of embedded connected devices where this requirement was not found in years past include set top boxes, DVR's, POS systems, both VoIP and mobile phones, connected home, and even automobile-based computing systems. Because wolfSSL supports the key embedded and real time operating systems, encryption standards, and authentication functionality, it is a natural choice for embedded systems developers to use when adding digital signature functionality.

Generally, the process for setting up code and file signing on an embedded device are as follows:

1. The embedded systems developer will generate an RSA key pair.
2. A server-side script-based tool is developed
    1. The server side tool will create a hash of the code to be loaded on the device (with SHA-256 for example).
    2. The hash is then digitally signed, also called RSA private encrypt.
    3. A package is created that contains the code along with the digital signature.
3. The package is loaded on the device along with a way to get the RSA public key. The hash is recreated on the device then digitally verified (also called RSA public decrypt) against the existing digital signature.

Benefits to enabling digital signatures on your device include:

1. Easily enable a secure method for allowing third parties to load files to your device.
2. Ensure against malicious files finding their way onto your device.
3. Digitally secure firmware updates
4. Ensure against firmware updates from unauthorized parties

General information on code signing:

https://en.wikipedia.org/wiki/Code_signing

COPYRIGHT ©2024 wolfSSL Inc.

13 OpenSSL Compatibility

13.1 Compatibility with OpenSSL

wolfSSL (formerly CyaSSL) provides an OpenSSL compatibility header, `wolfssl/openssl/ssl.h`, in addition to the wolfSSL native API, to ease the transition into using wolfSSL or to aid in porting an existing OpenSSL application over to wolfSSL. For an overview of the OpenSSL Compatibility Layer, please continue reading below. To view the complete set of OpenSSL functions supported by wolfSSL, please see the `wolfssl/openssl/ssl.h` file.

The OpenSSL Compatibility Layer maps a subset of the most commonly-used OpenSSL commands to wolfSSL's native API functions. This should allow for an easy replacement of OpenSSL by wolfSSL in your application or project without changing much code.

Our test beds for OpenSSL compatibility are stunnel and Lighttpd, which means that we build both of them with wolfSSL as a way to test our OpenSSL compatibility API.

Building wolfSSL With Compatibility Layer:

1. Enable with (`--enable-opensslextra`) or by defining the macro `OPENSSL_EXTRA`.

   ```
   ./configure --enable-opensslextra
   ```

2. Include <`wolfssl/options.h`> as first wolfSSL header
3. Header files for migration are located under:
   - `./wolfssl/openssl/*.h`
   - Ex: <`wolfssl/openssl/ssl.h`>

13.2 Differences Between wolfSSL and OpenSSL

Many people are curious how wolfSSL compares to OpenSSL and what benefits there are to using an SSL/TLS library that has been optimized to run on embedded platforms. Obviously, OpenSSL is free and presents no initial costs to begin using, but we believe that wolfSSL will provide you with more flexibility, an easier integration of SSL/TLS into your existing platform, current standards support, and much more – all provided under a very easy-to-use license model.

The points below outline several of the main differences between wolfSSL and OpenSSL.

1. With a 20-100 kB build size, wolfSSL is up to 20 times smaller than OpenSSL. wolfSSL is a better choice for resource constrained environments – where every byte matters.
2. wolfSSL is up to date with the most current standards of TLS 1.3 with DTLS. The wolfSSL team is dedicated to continually keeping wolfSSL up-to-date with current standards.
3. wolfSSL offers the best current ciphers and standards available today, including ciphers for streaming media support. In addition, the recently-introduced liboqs integration allows for you to start experimenting with post-quantum cryptography.
4. wolfSSL is dual licensed under both the GPLv3 as well as a commercial license, where OpenSSL is available only under their unique license from multiple sources.
5. wolfSSL is backed by an outstanding company who cares about its users and about their security, and is always willing to help. The team actively works to improve and expand wolfSSL. The wolfSSL team is based primarily out of Bozeman, MT, Portland, OR, and Seattle, WA, along with other team members located around the globe.
6. wolfSSL is the leading SSL/TLS library for real time, mobile, and embedded systems by virtue of its breadth of platform support and successful implementations on embedded environments. Chances are we've already been ported to your environment. If not, let us know and we'll be glad to help.

COPYRIGHT ©2024 wolfSSL Inc.

7. wolfSSL offers several abstraction layers to make integrating SSL into your environment and platform as easy as possible. With an OS layer, a custom I/O layer, and a C Standard Library abstraction layer, integration has never been so easy.
8. wolfSSL offers several support packages for wolfSSL. Available directly through phone, email or the wolfSSL product support forums, your questions are answered quickly and accurately to help you make progress on your project as quickly as possible.

13.3 Supported OpenSSL Structures

- SSL_METHOD holds SSL version information and is either a client or server method. (Same as WOLFSSL_METHOD in the native wolfSSL API).
- SSL_CTX holds context information including certificates. (Same as WOLFSSL_CTX in the native wolfSSL API).
- SSL holds session information for a secure connection. (Same as WOLFSSL in the native wolfSSL API).

13.4 Supported OpenSSL Functions

The three structures shown above are usually initialized in the following way:

```
SSL_METHOD* method = SSLv3_client_method();
SSL_CTX* ctx = SSL_CTX_new(method);
SSL* ssl = SSL_new(ctx);
```

This establishes a client side SSL version 3 method, creates a context based on the method, and initializes the SSL session with the context. A server side program is no different except that the SSL_METHOD is created using SSLv3_server_method(), or one of the available functions. For a list of supported functions, please see the Protocol Support section. When using the OpenSSL Compatibility layer, the functions in this section should be modified by removing the "wolf" prefix. For example, the native wolfSSL API function:

```
wolfTLSv1_client_method()
```

Becomes:

```
TLSv1_client_method()
```

When an SSL connection is no longer needed the following calls free the structures created during initialization.

```
SSL_CTX_free(ctx);
SSL_free(ssl);
```

SSL_CTX_free() has the additional responsibility of freeing the associated SSL_METHOD. Failing to use the XXX_free() functions will result in a resource leak. Using the system's free() instead of the SSL ones results in undefined behavior.

Once an application has a valid SSL pointer from SSL_new(), the SSL handshake process can begin. From the client's view, SSL_connect() will attempt to establish a secure connection.

```
SSL_set_fd(ssl, sockfd);
SSL_connect(ssl);
```

Before the SSL_connect() can be issued, the user must supply wolfSSL with a valid socket file descriptor, sockfd in the example above. sockfd is typically the result of the TCP function socket() which is later established using TCP connect(). The following creates a valid client side socket descriptor for use with a local wolfSSL server on port 11111, error handling is omitted for simplicity.

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);
sockaddr_in servaddr;
```

```
memset(&servaddr, 0, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(11111);
servaddr.sin_addr.s_addr = inet_addr("127.0.0.1");
connect(sockfd, (const sockaddr*)&servaddr, sizeof(servaddr));
```

Once a connection is established, the client may read and write to the server. Instead of using the TCP functions `send()` and `receive()`, wolfSSL and yaSSL use the SSL functions `SSL_write()` and `SSL_read()`. Here is a simple example from the client demo:

```
char msg[] = "hello wolfssl!";
int wrote = SSL_write(ssl, msg, sizeof(msg));
char reply[1024];
int read = SSL_read(ssl, reply, sizeof(reply));
reply[read] = 0;
printf("Server response: %s\n", reply);
```

The server connects in the same way, except that it uses `SSL_accept()` instead of `SSL_connect()`, analogous to the TCP API. See the server example for a complete server demo program.

13.5 x509 Certificates

Both the server and client can provide wolfSSL with certificates in either **PEM** or **DER**.

Typical usage is like this:

```
SSL_CTX_use_certificate_file(ctx, "certs/cert.pem",
SSL_FILETYPE_PEM);
SSL_CTX_use_PrivateKey_file(ctx, "certs/key.der",
SSL_FILETYPE_ASN1);
```

A key file can also be presented to the Context in either format. `SSL_FILETYPE_PEM` signifies the file is PEM formatted while `SSL_FILETYPE_ASN1` declares the file to be in DER format. To verify that the key file is appropriate for use with the certificate the following function can be used:

```
SSL_CTX_check_private_key(ctx);
```

14 Licensing

14.1 Open Source

wolfSSL, wolfCrypt, wolfMQTT, wolfTPM, wolfBoot, and wolfSentry are free software downloads and may be modified to the needs of the user as long as the user adheres to version three of the GPL License. The GPLv3 license can be found on the gnu.org website https://www.gnu.org/licenses/gpl.html.

wolfSSH software is a free software download and may be modified to the needs of the user as long as the user adheres to version three of the GPL license. The GPLv3 license can be found on the gnu.org website (https://www.gnu.org/licenses/gpl.html).

14.2 Commercial Licensing

Businesses and enterprises who wish to incorporate wolfSSL products into proprietary appliances or other commercial software products for re-distribution must license commercial versions. Commercial licenses for wolfSSL and wolfCrypt are available per end product or SKU. Licenses are generally issued for one product and include unlimited royalty-free distribution. Custom licensing terms are also available.

Commercial licenses are also available for wolfMQTT, wolfSSH, wolfTPM, wolfBoot, and wolfSentry. Please contact licensing@wolfssl.com with inquiries.

14.3 FIPS 140-2/3 Validation

wolfSSL is currently the leader in embedded FIPS certificates. For details on currently-active FIPS certificates and validation options, please see the wolfCrypt FIPS FAQ or contact fips@wolfssl.com.

14.4 Support Packages

Support packages for wolfSSL products are available on an annual basis directly from wolfSSL. With four different package options, you can compare them side-by-side and choose the package that best fits your specific needs. Please see our Support Packages page (https://www.wolfssl.com/products/support-and-maintenance) for more details.

COPYRIGHT ©2024 wolfSSL Inc.

15 Support and Consulting

15.1 How to Get Support

For general product support, wolfSSL (formerly CyaSSL) maintains an online forum for the wolfSSL product family. Please post to the forums or contact wolfSSL directly with any questions.

- wolfSSL (yaSSL) Forums: https://www.wolfssl.com/forums
- Email Support: support@wolfssl.com

For information regarding wolfSSL products, questions regarding licensing, or general comments, please contact wolfSSL by emailing facts@wolfssl.com. For support packages, please see Licensing.

15.1.1 Bugs Reports and Support Issues

If you are submitting a bug report or asking about a problem, please include the following information with your submission:

1. wolfSSL version number
2. Operating System version
3. Compiler version
4. The exact error you are seeing
5. A description of how we can reproduce or try to replicate this problem

With the above information, we will do our best to resolve your problems. Without this information, it is very hard to pinpoint the source of the problem. wolfSSL values your feedback and makes it a top priority to get back to you as soon as possible.

15.2 Consulting

wolfSSL offers both on and off site consulting - providing feature additions, porting, a Competitive Upgrade Program (see Competitive Upgrade Program), and design consulting.

15.2.1 Feature Additions and Porting

We can add additional features that you may need which are not currently offered in our products on a contract or co-development basis. We also offer porting services on our products to new host languages or new operating environments.

15.2.2 Competitive Upgrade Program

We will help you move from an outdated or expensive SSL/TLS library to wolfSSL with low cost and minimal disturbance to your code base.

Program Outline:

1. You need to currently be using a commercial competitor to wolfSSL.
2. You will receive up to one week of on-site consulting to switch out your old SSL library for wolfSSL. Travel expenses are not included.
3. Normally, up to one week is the right amount of time for us to make the replacement in your code and do initial testing. Additional consulting on a replacement is available as needed.
4. You will receive the standard wolfSSL royalty free license to ship with your product.

The purpose of this program is to enable users who are currently spending too much on their embedded SSL implementation to move to wolfSSL with ease. If you are interested in learning more, then please contact us at facts@wolfssl.com.

COPYRIGHT ©2024 wolfSSL Inc.

15.2.3 Design Consulting

If your application or framework needs to be secured with SSL/TLS but you are uncertain about how the optimal design of a secured system would be structured, we can help!

We offer design consulting for building SSL/TLS security into devices using wolfSSL. Our consultants can provide you with the following services:

1. *Assessment*: An evaluation of your current SSL/TLS implementation. We can give you advice on your current setup and how we think you could improve upon this by using wolfSSL.
2. *Design*: Looking at your system requirements and parameters, we'll work closely with you to make recommendations on how to implement wolfSSL into your application such that it provides you with optimal security.

If you would like to learn more about design consulting for building SSL into your application or device, please contact facts@wolfssl.com for more information.

16 wolfSSL (formerly CyaSSL) Updates

16.1 Product Release Information

We regularly post update information on Twitter. For additional release information, you can keep track of our projects on GitHub, follow us on Facebook, or follow our daily blog.

- wolfSSL on GitHub - https://www.github.com/wolfssl/wolfssl
- wolfSSL on Twitter - https://twitter.com/wolfSSL
- wolfSSL on Facebook - https://www.facebook.com/wolfSSL
- wolfSSL on Reddit - https://www.reddit.com/r/wolfssl/
- Daily Blog - https://www.wolfssl.com/blog

COPYRIGHT ©2024 wolfSSL Inc.

A wolfSSL API Reference

A.1 CertManager API

A.1.1 Functions

	Name
int	**wc_SignCert_cb**(int requestSz, int sType, byte * buf, word32 buffSz, int keyType, wc_SignCertCb signCb, void * signCtx, WC_RNG * rng)Sign a certificate or CSR using a callback function.
WOLFSSL_CERT_MANAGER *	**wolfSSL_CertManagerNew_ex**(void * heap)Allocates and initializes a new Certificate Manager context. This context may be used independent of SSL needs. It may be used to load certificates, verify certificates, and check the revocation status.
WOLFSSL_CERT_MANAGER *	**wolfSSL_CertManagerNew**(void)Allocates and initializes a new Certificate Manager context. This context may be used independent of SSL needs. It may be used to load certificates, verify certificates, and check the revocation status.
void	**wolfSSL_CertManagerFree**(WOLFSSL_CERT_MANAGER * cm)Frees all resources associated with the Certificate Manager context. Call this when you no longer need to use the Certificate Manager.
int	**wolfSSL_CertManagerLoadCA**(WOLFSSL_CERT_MANAGER * cm, const char * f, const char * d)Specifies the locations for CA certificate loading into the manager context. The PEM certificate CAfile may contain several trusted CA certificates. If CApath is not NULL it specifies a directory containing CA certificates in PEM format.
int	**wolfSSL_CertManagerLoadCABuffer**(WOLFSSL_CERT_MANAGER * cm, const unsigned char * buff, long sz, int format)Loads the CA Buffer by calling wolfSSL_CTX_load_verify_buffer and returning that result using a temporary cm so as not to lose the information in the cm passed into the function.
int	**wolfSSL_CertManagerUnloadCAs**(WOLFSSL_CERT_MANAGER * cm)This function unloads the CA signer list.
int	**wolfSSL_CertManagerUnloadIntermediateCerts**(WOLFSSL_CE * cm)This function unloads intermediate certificates add to the CA signer list.
int	**wolfSSL_CertManagerUnload_trust_peers**(WOLFSSL_CERT_MA * cm)The function will free the Trusted Peer linked list and unlocks the trusted peer list.
int	**wolfSSL_CertManagerVerify**(WOLFSSL_CERT_MANAGER * cm, const char * f, int format)Specifies the certificate to verify with the Certificate Manager context. The format can be SSL_FILETYPE_PEM or SSL_FILETYPE_ASN1.

	Name
int	**wolfSSL_CertManagerVerifyBuffer**(WOLFSSL_CERT_MANAGER * cm, const unsigned char * buff, long sz, int format)Specifies the certificate buffer to verify with the Certificate Manager context. The format can be SSL_FILETYPE_PEM or SSL_FILETYPE_ASN1.
void	**wolfSSL_CertManagerSetVerify**(WOLFSSL_CERT_MANAGER * cm, VerifyCallback verify_callback)The function sets the verifyCallback function in the Certificate Manager. If present, it will be called for each cert loaded. If there is a verification error, the verify callback can be used to over-ride the error.
int	**wolfSSL_CertManagerEnableCRL**(WOLFSSL_CERT_MANAGER * cm, int options)Turns on Certificate Revocation List checking when verifying certificates with the Certificate Manager. By default, CRL checking is off. options include WOLFSSL_CRL_CHECKALL which performs CRL checking on each certificate in the chain versus the Leaf certificate only which is the default.
int	**wolfSSL_CertManagerDisableCRL**(WOLFSSL_CERT_MANAGER * cm)Turns off Certificate Revocation List checking when verifying certificates with the Certificate Manager. By default, CRL checking is off. You can use this function to temporarily or permanently disable CRL checking with this Certificate Manager context that previously had CRL checking enabled.
int	**wolfSSL_CertManagerLoadCRL**(WOLFSSL_CERT_MANAGER * cm, const char * path, int type, int monitor)Error checks and passes through to LoadCRL() in order to load the cert into the CRL for revocation checking. An updated CRL can be loaded by first calling wolfSSL_CertManagerFreeCRL, then loading the new CRL.
int	**wolfSSL_CertManagerLoadCRLBuffer**(WOLFSSL_CERT_MANAGE * cm, const unsigned char * buff, long sz, int type)The function loads the CRL file by calling BufferLoadCRL.
int	**wolfSSL_CertManagerSetCRL_Cb**(WOLFSSL_CERT_MANAGER * cm, CbMissingCRL cb)This function sets the CRL Certificate Manager callback. If HAVE_CRL is defined and a matching CRL record is not found then the cbMissingCRL is called (set via wolfSSL_CertManagerSetCRL_Cb). This allows you to externally retrieve the CRL and load it.

	Name
int	**wolfSSL_CertManagerSetCRLUpdate_Cb**(WOLFSSL_CERT_MANA * cm, CbUpdateCRL cb)This function sets the CRL Update callback. If HAVE_CRL and HAVE_CRL_UPDATE_CB is defined , and an entry with the same issuer and a lower CRL number exists when a CRL is added, then the CbUpdateCRL is called with the details of the existing entry and the new one replacing it.
int	**wolfSSL_CertManagerGetCRLInfo**(WOLFSSL_CERT_MANAGER * cm, CrlInfo * info, const byte * buff, long sz, int type)This function yields a structure with parsed CRL information from an encoded CRL buffer.
int	**wolfSSL_CertManagerCheckOCSP**(WOLFSSL_CERT_MANAGER * cm, const unsigned char * der, int sz)The function enables the WOLFSSL_CERT_MANAGER's member, ocspEnabled to signify that the OCSP check option is enabled.
int	**wolfSSL_CertManagerEnableOCSP**(WOLFSSL_CERT_MANAGER * cm, int options)Turns on OCSP if it's turned off and if compiled with the set option available.
int	**wolfSSL_CertManagerDisableOCSP**(WOLFSSL_CERT_MANAGER * cm)Disables OCSP certificate revocation.
int	**wolfSSL_CertManagerSetOCSPOverrideURL**(WOLFSSL_CERT_M * cm, const char * url)The function copies the url to the ocspOverrideURL member of the WOLFSSL_CERT_MANAGER structure.
int	**wolfSSL_CertManagerSetOCSP_Cb**(WOLFSSL_CERT_MANAGER * cm, CbOCSPIO ioCb, CbOCSPRespFree respFreeCb, void * ioCbCtx)The function sets the OCSP callback in the WOLFSSL_CERT_MANAGER.
int	**wolfSSL_CertManagerEnableOCSPStapling**(WOLFSSL_CERT_MA * cm)This function turns on OCSP stapling if it is not turned on as well as set the options.

A.1.2 Functions Documentation

```
int wc_SignCert_cb(
    int requestSz,
    int sType,
    byte * buf,
    word32 buffSz,
    int keyType,
    wc_SignCertCb signCb,
    void * signCtx,
    WC_RNG * rng
)
```

Sign a certificate or CSR using a callback function.

Parameters:

- **requestSz** Size of the certificate body to sign (from Cert.bodySz).
- **sType** Signature algorithm type (e.g., CTC_SHA256wRSA, CTC_SHA256wECDSA).
- **buf** Buffer containing the certificate/CSR DER data to sign.
- **buffSz** Total size of the buffer (must be large enough for signature).
- **keyType** Type of key used for signing. Only RSA_TYPE and ECC_TYPE are supported.
- **signCb** User-provided signing callback function.
- **signCtx** Context pointer passed to the signing callback.
- **rng** Random number generator (may be NULL if not needed).

See:

- wc_SignCertCb
- wc_SignCert
- wc_SignCert_ex
- wc_MakeCert
- wc_MakeCertReq

Return:

- Size of the signed certificate/CSR on success.
- BAD_FUNC_ARG if signCb or buf is NULL, buffSz is 0, or keyType is not RSA_TYPE or ECC_TYPE.
- BUFFER_E if the buffer is too small for the signed certificate.
- MEMORY_E if memory allocation fails.
- Negative error code on other failures.

This function signs a certificate or Certificate Signing Request (CSR) using a user-provided signing callback. This allows external signing implementations (e.g., TPM, HSM) without requiring the crypto callback infrastructure, making it suitable for FIPS-compliant applications.

The function performs the following:

1. Hashes the certificate/CSR body according to the signature algorithm
2. Encodes the hash (RSA) or prepares it for signing (ECC)
3. Calls the user-provided callback to perform the actual signing
4. Encodes the signature into the certificate/CSR DER structure

NOTE: Only RSA and ECC key types are supported. Ed25519, Ed448, and post-quantum algorithms (Falcon, Dilithium, SLH-DSA) sign messages directly rather than hashes, so they cannot use this callback-based API. Use wc_SignCert_ex for those algorithms.

NOTE: This function does NOT support async crypto (WOLFSSL_ASYNC_CRYPT). The internal context is local to this function and cannot persist across async re-entry.

Example

```
Cert cert;
byte derBuf[4096];
int derSz;
MySignCtx myCtx;

wc_InitCert(&cert);
```

```
derSz = wc_MakeCert(&cert, derBuf, sizeof(derBuf), NULL, NULL, &rng);

derSz = wc_SignCert_cb(cert.bodySz, cert.sigType, derBuf, sizeof(derBuf),
                       RSA_TYPE, mySignCallback, &myCtx, &rng);
if (derSz > 0) {
    printf("Signed certificate is %d bytes\n", derSz);
}
```

```
WOLFSSL_CERT_MANAGER * wolfSSL_CertManagerNew_ex(
    void * heap
)
```

Allocates and initializes a new Certificate Manager context. This context may be used independent of SSL needs. It may be used to load certificates, verify certificates, and check the revocation status.

Parameters:

- **heap** pointer to a heap hint for memory allocation.

See: wolfSSL_CertManagerFree

Return:

- WOLFSSL_CERT_MANAGER If successful the call will return a valid WOLFSSL_CERT_MANAGER pointer.
- NULL will be returned for an error state.

```
WOLFSSL_CERT_MANAGER * wolfSSL_CertManagerNew(
    void
)
```

Allocates and initializes a new Certificate Manager context. This context may be used independent of SSL needs. It may be used to load certificates, verify certificates, and check the revocation status.

Parameters:

- **none** No parameters.

See: wolfSSL_CertManagerFree

Return:

- WOLFSSL_CERT_MANAGER If successful the call will return a valid WOLFSSL_CERT_MANAGER pointer.
- NULL will be returned for an error state.

Example

```
#import <wolfssl/ssl.h>

WOLFSSL_CERT_MANAGER* cm;
cm = wolfSSL_CertManagerNew();
if (cm == NULL) {
// error creating new cert manager
}
```

```
void wolfSSL_CertManagerFree(
    WOLFSSL_CERT_MANAGER * cm
)
```

Frees all resources associated with the Certificate Manager context. Call this when you no longer need to use the Certificate Manager.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().

See: wolfSSL_CertManagerNew

Return: none

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CERT_MANAGER* cm;
...
wolfSSL_CertManagerFree(cm);
```

```
int wolfSSL_CertManagerLoadCA(
    WOLFSSL_CERT_MANAGER * cm,
    const char * f,
    const char * d
)
```

Specifies the locations for CA certificate loading into the manager context. The PEM certificate CAfile may contain several trusted CA certificates. If CApath is not NULL it specifies a directory containing CA certificates in PEM format.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().
- **file** pointer to the name of the file containing CA certificates to load.
- **path** pointer to the name of a directory path containing CA c ertificates to load. The NULL pointer may be used if no certificate directory is desired.

See: wolfSSL_CertManagerVerify

Return:

- SSL_SUCCESS If successful the call will return.
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- BAD_FUNC_ARG is the error that will be returned if a pointer is not provided.
- SSL_FATAL_ERROR - will be returned upon failure.

Example

```
#include <wolfssl/ssl.h>

int ret = 0;
WOLFSSL_CERT_MANAGER* cm;
...
ret = wolfSSL_CertManagerLoadCA(cm, “path/to/cert-file.pem”, 0);
if (ret != SSL_SUCCESS) {
// error loading CA certs into cert manager
}
```

```
int wolfSSL_CertManagerLoadCABuffer(
    WOLFSSL_CERT_MANAGER * cm,
    const unsigned char * buff,
    long sz,
    int format
)
```

Loads the CA Buffer by calling wolfSSL_CTX_load_verify_buffer and returning that result using a temporary cm so as not to lose the information in the cm passed into the function.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().
- **in** buffer for cert information.
- **sz** length of the buffer.
- **format** certificate format, either PEM or DER.

See:

- wolfSSL_CTX_load_verify_buffer
- ProcessChainBuffer
- ProcessBuffer
- cm_pick_method

Return:

- SSL_FATAL_ERROR is returned if the WOLFSSL_CERT_MANAGER struct is NULL or if wolfSSL_CTX_new() returns NULL.
- SSL_SUCCESS is returned for a successful execution.

Example

```
WOLFSSL_CERT_MANAGER* cm = (WOLFSSL_CERT_MANAGER*)vp;
…
const unsigned char* in;
long sz;
int format;
…
if(wolfSSL_CertManagerLoadCABuffer(vp, sz, format) != SSL_SUCCESS){
    Error returned. Failure case code block.
}
```

```
int wolfSSL_CertManagerUnloadCAs(
    WOLFSSL_CERT_MANAGER * cm
)
```

This function unloads the CA signer list.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().

See: UnlockMutex

Return:

- SSL_SUCCESS returned on successful execution of the function.
- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER is NULL.
- BAD_MUTEX_E returned if there was a mutex error.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = wolfSSL_CTX_new(protocol method);
WOLFSSL_CERT_MANAGER* cm = wolfSSL_CTX_GetCertManager(ctx);
...
if(wolfSSL_CertManagerUnloadCAs(cm) != SSL_SUCCESS){
    Failure case.
}
```

```
int wolfSSL_CertManagerUnloadIntermediateCerts(
    WOLFSSL_CERT_MANAGER * cm
)
```

This function unloads intermediate certificates add to the CA signer list.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().

See: UnlockMutex

Return:

- SSL_SUCCESS returned on successful execution of the function.
- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER is NULL.
- BAD_MUTEX_E returned if there was a mutex error.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = wolfSSL_CTX_new(protocol method);
WOLFSSL_CERT_MANAGER* cm = wolfSSL_CTX_GetCertManager(ctx);
...
if(wolfSSL_CertManagerUnloadIntermediateCerts(cm) != SSL_SUCCESS){
    Failure case.
}
```

```
int wolfSSL_CertManagerUnload_trust_peers(
    WOLFSSL_CERT_MANAGER * cm
)
```

The function will free the Trusted Peer linked list and unlocks the trusted peer list.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().

See: UnLockMutex

Return:

- SSL_SUCCESS if the function completed normally.
- BAD_FUNC_ARG if the WOLFSSL_CERT_MANAGER is NULL.
- BAD_MUTEX_E mutex error if tpLock, a member of the WOLFSSL_CERT_MANAGER struct, is 0 (nill).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(Protocol define);
WOLFSSL_CERT_MANAGER* cm = wolfSSL_CertManagerNew();
...
if(wolfSSL_CertManagerUnload_trust_peers(cm) != SSL_SUCCESS){
    The function did not execute successfully.
}
```

```
int wolfSSL_CertManagerVerify(
    WOLFSSL_CERT_MANAGER * cm,
    const char * f,
    int format
)
```

Specifies the certificate to verify with the Certificate Manager context. The format can be SSL_FILETYPE_PEM or SSL_FILETYPE_ASN1.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().
- **fname** pointer to the name of the file containing the certificates to verify.
- **format** format of the certificate to verify - either SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CertManagerLoadCA
- wolfSSL_CertManagerVerifyBuffer

Return:

- SSL_SUCCESS If successful.
- ASN_SIG_CONFIRM_E will be returned if the signature could not be verified.
- ASN_SIG_OID_E will be returned if the signature type is not supported.
- CRL_CERT_REVOKED is an error that is returned if this certificate has been revoked.
- CRL_MISSING is an error that is returned if a current issuer CRL is not available.
- ASN_BEFORE_DATE_E will be returned if the current date is before the before date.
- ASN_AFTER_DATE_E will be returned if the current date is after the after date.
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- BAD_FUNC_ARG is the error that will be returned if a pointer is not provided.

Example

```
int ret = 0;
WOLFSSL_CERT_MANAGER* cm;
...

ret = wolfSSL_CertManagerVerify(cm, “path/to/cert-file.pem”,
SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    error verifying certificate
}
```

```
int wolfSSL_CertManagerVerifyBuffer(
    WOLFSSL_CERT_MANAGER * cm,
    const unsigned char * buff,
```

```
    long sz,
    int format
)
```

Specifies the certificate buffer to verify with the Certificate Manager context. The format can be SSL_FILETYPE_PEM or SSL_FILETYPE_ASN1.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().
- **buff** buffer containing the certificates to verify.
- **sz** size of the buffer, buf.
- **format** format of the certificate to verify, located in buf - either SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CertManagerLoadCA
- wolfSSL_CertManagerVerify

Return:

- SSL_SUCCESS If successful.
- ASN_SIG_CONFIRM_E will be returned if the signature could not be verified.
- ASN_SIG_OID_E will be returned if the signature type is not supported.
- CRL_CERT_REVOKED is an error that is returned if this certificate has been revoked.
- CRL_MISSING is an error that is returned if a current issuer CRL is not available.
- ASN_BEFORE_DATE_E will be returned if the current date is before the before date.
- ASN_AFTER_DATE_E will be returned if the current date is after the after date.
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- BAD_FUNC_ARG is the error that will be returned if a pointer is not provided.

Example

```
#include <wolfssl/ssl.h>

int ret = 0;
int sz = 0;
WOLFSSL_CERT_MANAGER* cm;
byte certBuff[...];
...

ret = wolfSSL_CertManagerVerifyBuffer(cm, certBuff, sz, SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    error verifying certificate
}
```

```
void wolfSSL_CertManagerSetVerify(
    WOLFSSL_CERT_MANAGER * cm,
    VerifyCallback verify_callback
)
```

The function sets the verifyCallback function in the Certificate Manager. If present, it will be called for each cert loaded. If there is a verification error, the verify callback can be used to over-ride the error.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().
- **verify_callback** a VerifyCallback function pointer to the callback routine

See: wolfSSL_CertManagerVerify

Return: none No return.

Example

```
#include <wolfssl/ssl.h>

int myVerify(int preverify, WOLFSSL_X509_STORE_CTX* store)
{ // do custom verification of certificate }

WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(Protocol define);
WOLFSSL_CERT_MANAGER* cm = wolfSSL_CertManagerNew();
...
wolfSSL_CertManagerSetVerify(cm, myVerify);
```

```
int wolfSSL_CertManagerEnableCRL(
    WOLFSSL_CERT_MANAGER * cm,
    int options
)
```

Turns on Certificate Revocation List checking when verifying certificates with the Certificate Manager. By default, CRL checking is off. options include WOLFSSL_CRL_CHECKALL which performs CRL checking on each certificate in the chain versus the Leaf certificate only which is the default.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().
- **options** options to use when enabling the Certification Manager, cm.

See: wolfSSL_CertManagerDisableCRL

Return:

- SSL_SUCCESS If successful the call will return.
- NOT_COMPILED_IN will be returned if wolfSSL was not built with CRL enabled.
- MEMORY_E will be returned if an out of memory condition occurs.
- BAD_FUNC_ARG is the error that will be returned if a pointer is not provided.
- SSL_FAILURE will be returned if the CRL context cannot be initialized properly.

Example

```
#include <wolfssl/ssl.h>

int ret = 0;
WOLFSSL_CERT_MANAGER* cm;
...

ret = wolfSSL_CertManagerEnableCRL(cm, 0);
if (ret != SSL_SUCCESS) {
    error enabling cert manager
}

...
```

```
int wolfSSL_CertManagerDisableCRL(
    WOLFSSL_CERT_MANAGER * cm
)
```

Turns off Certificate Revocation List checking when verifying certificates with the Certificate Manager. By default, CRL checking is off. You can use this function to temporarily or permanently disable CRL checking with this Certificate Manager context that previously had CRL checking enabled.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().

See: wolfSSL_CertManagerEnableCRL

Return:

- SSL_SUCCESS If successful the call will return.
- BAD_FUNC_ARG is the error that will be returned if a function pointer is not provided.

Example

```
#include <wolfssl/ssl.h>

int ret = 0;
WOLFSSL_CERT_MANAGER* cm;
...
ret = wolfSSL_CertManagerDisableCRL(cm);
if (ret != SSL_SUCCESS) {
    error disabling cert manager
}
...
```

```
int wolfSSL_CertManagerLoadCRL(
    WOLFSSL_CERT_MANAGER * cm,
    const char * path,
    int type,
    int monitor
)
```

Error checks and passes through to LoadCRL() in order to load the cert into the CRL for revocation checking. An updated CRL can be loaded by first calling wolfSSL_CertManagerFreeCRL, then loading the new CRL.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().
- **path** a constant char pointer holding the CRL path.
- **type** type of certificate to be loaded.
- **monitor** requests monitoring in LoadCRL().

See:

- wolfSSL_CertManagerEnableCRL
- wolfSSL_LoadCRL
- wolfSSL_CertManagerFreeCRL

Return:

- SSL_SUCCESS if there is no error in wolfSSL_CertManagerLoadCRL and if LoadCRL returns successfully.
- BAD_FUNC_ARG if the WOLFSSL_CERT_MANAGER struct is NULL.
- SSL_FATAL_ERROR if wolfSSL_CertManagerEnableCRL returns anything other than SSL_SUCCESS.
- BAD_PATH_ERROR if the path is NULL.
- MEMORY_E if LoadCRL fails to allocate heap memory.

Example

```
#include <wolfssl/ssl.h>

int wolfSSL_LoadCRL(WOLFSSL* ssl, const char* path, int type,
int monitor);
…
wolfSSL_CertManagerLoadCRL(SSL_CM(ssl), path, type, monitor);
```

```
int wolfSSL_CertManagerLoadCRLBuffer(
    WOLFSSL_CERT_MANAGER * cm,
    const unsigned char * buff,
    long sz,
    int type
)
```

The function loads the CRL file by calling BufferLoadCRL.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure.
- **buff** a constant byte type and is the buffer.
- **sz** a long int representing the size of the buffer.
- **type** a long integer that holds the certificate type.

See:

- BufferLoadCRL
- wolfSSL_CertManagerEnableCRL

Return:

- SSL_SUCCESS returned if the function completed without errors.
- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER is NULL.
- SSL_FATAL_ERROR returned if there is an error associated with the WOLFSSL_CERT_MANAGER.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CERT_MANAGER* cm;
const unsigned char* buff;
long sz; size of buffer
int type;  cert type
...
int ret = wolfSSL_CertManagerLoadCRLBuffer(cm, buff, sz, type);
if(ret == SSL_SUCCESS){
return ret;
} else {
    Failure case.
}
```

```
int wolfSSL_CertManagerSetCRL_Cb(
    WOLFSSL_CERT_MANAGER * cm,
    CbMissingCRL cb
)
```

This function sets the CRL Certificate Manager callback. If HAVE_CRL is defined and a matching CRL record is not found then the cbMissingCRL is called (set via wolfSSL_CertManagerSetCRL_Cb). This allows you to externally retrieve the CRL and load it.

Parameters:

- **cm** the WOLFSSL_CERT_MANAGER structure holding the information for the certificate.
- **cb** a function pointer to (*CbMissingCRL) that is set to the cbMissingCRL member of the WOLFSSL_CERT_MANAGER.

See:

- CbMissingCRL
- wolfSSL_SetCRL_Cb

Return:

- SSL_SUCCESS returned upon successful execution of the function and subroutines.
- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER structure is NULL.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = wolfSSL_CTX_new(protocol method);
WOLFSSL* ssl = wolfSSL_new(ctx);
…
void cb(const char* url){
    Function body.
}
…
CbMissingCRL cb = CbMissingCRL;
…
if(ctx){
    return wolfSSL_CertManagerSetCRL_Cb(SSL_CM(ssl), cb);
}
```

```
int wolfSSL_CertManagerSetCRLUpdate_Cb(
    WOLFSSL_CERT_MANAGER * cm,
    CbUpdateCRL cb
)
```

This function sets the CRL Update callback. If HAVE_CRL and HAVE_CRL_UPDATE_CB is defined , and an entry with the same issuer and a lower CRL number exists when a CRL is added, then the CbUpdateCRL is called with the details of the existing entry and the new one replacing it.

Parameters:

- **cm** the WOLFSSL_CERT_MANAGER structure holding the information for the certificate.
- **cb** a function pointer to (*CbUpdateCRL) that is set to the cbUpdateCRL member of the WOLFSSL_CERT_MANAGER. Signature requirement: void (*CbUpdateCRL)(CrlInfo *old, CrlInfo* new);

See: CbUpdateCRL

Return:

- SSL_SUCCESS returned upon successful execution of the function and subroutines.
- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER structure is NULL.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = wolfSSL_CTX_new(protocol method);
WOLFSSL* ssl = wolfSSL_new(ctx);
…
void cb(CrlInfo *old, CrlInfo *new){
    Function body.
}
…
CbUpdateCRL cb = CbUpdateCRL;
…
if(ctx){
    return wolfSSL_CertManagerSetCRLUpdate_Cb(SSL_CM(ssl), cb);
}
```

```
int wolfSSL_CertManagerGetCRLInfo(
    WOLFSSL_CERT_MANAGER * cm,
    CrlInfo * info,
    const byte * buff,
    long sz,
    int type
)
```

This function yields a structure with parsed CRL information from an encoded CRL buffer.

Parameters:

- **cm** the WOLFSSL_CERT_MANAGER structure..
- **info** pointer to caller managed CrlInfo structure that will receive the CRL information.
- **buff** input buffer containing encoded CRL.
- **sz** the length in bytes of the input CRL data in buff.
- **type** WOLFSSL_FILETYPE_PEM or WOLFSSL_FILETYPE_DER
- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().

See:

- CbUpdateCRL
- wolfSSL_SetCRL_Cb
- wolfSSL_CertManagerLoadCRL

Return:

- SSL_SUCCESS returned upon successful execution of the function and subroutines.
- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER structure is NULL.
- SSL_SUCCESS returned upon successful execution of the function and subroutines.
- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER structure is NULL.

Example

```
#include <wolfssl/ssl.h>

CrlInfo info;
WOLFSSL_CERT_MANAGER* cm = NULL;

cm = wolfSSL_CertManagerNew();

// Read crl data from file into buffer

wolfSSL_CertManagerGetCRLInfo(cm, &info, crlData, crlDataLen,
                              WOLFSSL_FILETYPE_PEM);
```

This function frees the CRL stored in the Cert Manager. An application can update the CRL by calling wolfSSL_CertManagerFreeCRL and then loading the new CRL.

Example

```
#include <wolfssl/ssl.h>

const char* crl1     = "./certs/crl/crl.pem";
WOLFSSL_CERT_MANAGER* cm = NULL;

cm = wolfSSL_CertManagerNew();
wolfSSL_CertManagerLoadCRL(cm, crl1, WOLFSSL_FILETYPE_PEM, 0);
…
wolfSSL_CertManagerFreeCRL(cm);
```

```
int wolfSSL_CertManagerCheckOCSP(
    WOLFSSL_CERT_MANAGER * cm,
    const unsigned char * der,
    int sz
)
```

The function enables the WOLFSSL_CERT_MANAGER's member, ocspEnabled to signify that the OCSP check option is enabled.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().
- **der** a byte pointer to the certificate.
- **sz** an int type representing the size of the DER cert.

See:

- ParseCertRelative
- CheckCertOCSP

Return:

- SSL_SUCCESS returned on successful execution of the function. The ocspEnabled member of the WOLFSSL_CERT_MANAGER is enabled.
- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER structure is NULL or if an argument value that is not allowed is passed to a subroutine.
- MEMORY_E returned if there is an error allocating memory within this function or a subroutine.

Example

```
#import <wolfssl/ssl.h>

WOLFSSL* ssl = wolfSSL_new(ctx);
byte* der;
int sz; size of der
...
if(wolfSSL_CertManagerCheckOCSP(cm, der, sz) != SSL_SUCCESS){
 Failure case.
}
```

```
int wolfSSL_CertManagerEnableOCSP(
    WOLFSSL_CERT_MANAGER * cm,
    int options
)
```

Turns on OCSP if it's turned off and if compiled with the set option available.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().
- **options** used to set values in WOLFSSL_CERT_MANAGER struct.

See: wolfSSL_CertManagerNew

Return:

- SSL_SUCCESS returned if the function call is successful.
- BAD_FUNC_ARG if cm struct is NULL.
- MEMORY_E if WOLFSSL_OCSP struct value is NULL.
- SSL_FAILURE initialization of WOLFSSL_OCSP struct fails to initialize.
- NOT_COMPILED_IN build not compiled with correct feature enabled.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = wolfSSL_CTX_new(protocol method);
WOLFSSL* ssl = wolfSSL_new(ctx);
WOLFSSL_CERT_MANAGER* cm = wolfSSL_CertManagerNew();
int options;
…
```

```
if(wolfSSL_CertManagerEnableOCSP(SSL_CM(ssl), options) != SSL_SUCCESS){
    Failure case.
}
```

```
int wolfSSL_CertManagerDisableOCSP(
    WOLFSSL_CERT_MANAGER * cm
)
```

Disables OCSP certificate revocation.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure.

See: wolfSSL_DisableCRL

Return:

- SSL_SUCCESS wolfSSL_CertMangerDisableCRL successfully disabled the crlEnabled member of the WOLFSSL_CERT_MANAGER structure.
- BAD_FUNC_ARG the WOLFSSL structure was NULL.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = wolfSSL_CTX_new(method);
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_CertManagerDisableOCSP(ssl) != SSL_SUCCESS){
    Fail case.
}
```

```
int wolfSSL_CertManagerSetOCSPOverrideURL(
    WOLFSSL_CERT_MANAGER * cm,
    const char * url
)
```

The function copies the url to the ocspOverrideURL member of the WOLFSSL_CERT_MANAGER structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- ocspOverrideURL

- wolfSSL_SetOCSP_OverrideURL

Return:

- SSL_SUCCESS the function was able to execute as expected.
- BAD_FUNC_ARG the WOLFSSL_CERT_MANAGER struct is NULL.
- MEMEORY_E Memory was not able to be allocated for the ocspOverrideURL member of the certificate manager.

Example

```
#include <wolfssl/ssl.h>
WOLFSSL_CERT_MANAGER* cm = wolfSSL_CertManagerNew();
const char* url;
…
int wolfSSL_SetOCSP_OverrideURL(WOLFSSL* ssl, const char* url)
…
if(wolfSSL_CertManagerSetOCSPOverrideURL(SSL_CM(ssl), url) != SSL_SUCCESS){
    Failure case.
}
```

```
int wolfSSL_CertManagerSetOCSP_Cb(
    WOLFSSL_CERT_MANAGER * cm,
    CbOCSPIO ioCb,
    CbOCSPRespFree respFreeCb,
    void * ioCbCtx
)
```

The function sets the OCSP callback in the WOLFSSL_CERT_MANAGER.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure.
- **ioCb** a function pointer of type CbOCSPIO.
- **respFreeCb** - a function pointer of type CbOCSPRespFree.
- **ioCbCtx** - a void pointer variable to the I/O callback user registered context.

See:

- wolfSSL_CertManagerSetOCSPOverrideURL
- wolfSSL_CertManagerCheckOCSP
- wolfSSL_CertManagerEnableOCSPStapling
- wolfSSL_EnableOCSP
- wolfSSL_DisableOCSP
- wolfSSL_SetOCSP_Cb

Return:

- SSL_SUCCESS returned on successful execution. The arguments are saved in the WOLFSSL_CERT_MANAGER structure.

- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER is NULL.

Example

```
#include <wolfssl/ssl.h>

wolfSSL_SetOCSP_Cb(WOLFSSL* ssl, CbOCSPIO ioCb,
CbOCSPRespFree respFreeCb, void* ioCbCtx){
…
return wolfSSL_CertManagerSetOCSP_Cb(SSL_CM(ssl), ioCb, respFreeCb, ioCbCtx);
```

```
int wolfSSL_CertManagerEnableOCSPStapling(
    WOLFSSL_CERT_MANAGER * cm
)
```

This function turns on OCSP stapling if it is not turned on as well as set the options.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, a member of the WOLFSSL_CTX structure.

See: wolfSSL_CTX_EnableOCSPStapling

Return:

- SSL_SUCCESS returned if there were no errors and the function executed successfully.
- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER structure is NULL or otherwise if there was a unpermitted argument value passed to a subroutine.
- MEMORY_E returned if there was an issue allocating memory.
- SSL_FAILURE returned if the initialization of the OCSP structure failed.
- NOT_COMPILED_IN returned if wolfSSL was not compiled with HAVE_CERTIFICATE_STATUS_REQUEST option.

Example

```
int wolfSSL_CTX_EnableOCSPStapling(WOLFSSL_CTX* ctx){
…
return wolfSSL_CertManagerEnableOCSPStapling(ctx->cm);
```

A.2 Memory Handling

A.1.2.25 function wolfSSL_CertManagerEnableOCSPStapling

A.2.1 Functions

	Name
void *	**wolfSSL_Malloc**(size_t size, void * heap, int type)This function is similar to malloc(), but calls the memory allocation function which wolfSSL has been configured to use. By default, wolfSSL uses malloc(). This can be changed using the wolfSSL memory abstraction layer _ see wolfSSL_SetAllocators(). Note wolfSSL_Malloc is not called directly by wolfSSL, but instead called by macro XMALLOC. For the default build only the size argument exists. If using WOLFSSL_STATIC_MEMORY build then heap and type arguments are included.
void	**wolfSSL_Free**(void * ptr, void * heap, int type)This function is similar to free(), but calls the memory free function which wolfSSL has been configured to use. By default, wolfSSL uses free(). This can be changed using the wolfSSL memory abstraction layer _ see wolfSSL_SetAllocators(). Note wolfSSL_Free is not called directly by wolfSSL, but instead called by macro XFREE. For the default build only the ptr argument exists. If using WOLFSSL_STATIC_MEMORY build then heap and type arguments are included.
void *	**wolfSSL_Realloc**(void * ptr, size_t size, void * heap, int type)This function is similar to realloc(), but calls the memory re_allocation function which wolfSSL has been configured to use. By default, wolfSSL uses realloc(). This can be changed using the wolfSSL memory abstraction layer _ see wolfSSL_SetAllocators(). Note wolfSSL_Realloc is not called directly by wolfSSL, but instead called by macro XREALLOC. For the default build only the size argument exists. If using WOLFSSL_STATIC_MEMORY build then heap and type arguments are included.
int	**wolfSSL_SetAllocators**(wolfSSL_Malloc_cb mf, wolfSSL_Free_cb ff, wolfSSL_Realloc_cb rf)This function registers the allocation functions used by wolfSSL. By default, if the system supports it, malloc/free and realloc are used. Using this function allows the user at runtime to install their own memory handlers.

	Name
int	**wolfSSL_StaticBufferSz**(byte * buffer, word32 sz, int flag)This function is available when static memory feature is used (–enable_staticmemory). It gives the optimum buffer size for memory “buckets”. This allows for a way to compute buffer size so that no extra unused memory is left at the end after it has been partitioned. For the none _ex version of this function the default bucket and distribution list set during compile time is used. The returned value, if positive, is the computed buffer size to use.
int	**wolfSSL_MemoryPaddingSz**(void)This function is available when static memory feature is used (–enable_staticmemory). It gives the size of padding needed for each partition of memory. This padding size will be the size needed to contain a memory management structure along with any extra for memory alignment.
int	**wolfSSL_CTX_load_static_memory**(WOLFSSL_CTX ** ctx, wolfSSL_method_func method, unsigned char * buf, unsigned int sz, int flag, int max)This function is used to set aside static memory for a CTX. Memory set aside is then used for the CTX’s lifetime and for any SSL objects created from the CTX. By passing in a NULL ctx pointer and a wolfSSL_method_func function the creation of the CTX itself will also use static memory. wolfSSL_method_func has the function signature of WOLFSSL_METHOD* (*wolfSSL_method_func)(void* heap);. Passing in 0 for max makes it behave as if not set and no max concurrent use restrictions is in place. The flag value passed in determines how the memory is used and behavior while operating. Available flags are the following.
int	**wolfSSL_CTX_is_static_memory**(WOLFSSL_CTX * ctx, WOLFSSL_MEM_STATS * mem_stats)This function does not change any of the connections behavior and is used only for gathering information about the static memory usage.

	Name
int	**wolfSSL_is_static_memory**(WOLFSSL * ssl, WOLFSSL_MEM_CONN_STATS * mem_stats)wolfSSL_is_static_memory is used to gather information about a SSL's static memory usage. The return value indicates if static memory is being used and WOLFSSL_MEM_CONN_STATS will be filled out if and only if the flag WOLFMEM_TRACK_STATS was passed to the parent CTX when loading in static memory.
int	**wc_LoadStaticMemory**(WOLFSSL_HEAP_HINT ** pHint, unsigned char * buf, unsigned int sz, int flag, int max)This function is used to set aside static memory for wolfCrypt use. Memory can be used by passing the created heap hint into functions. An example of this is when calling wc_InitRng_ex. The flag value passed in determines how the memory is used and behavior while operating, in general wolfCrypt operations will use memory from a WOLFMEM_GENERAL pool. Available flags are the following.
int	**wc_LoadStaticMemory_ex**(WOLFSSL_HEAP_HINT ** pHint, unsigned int listSz, const word32 * sizeList, const word32 * distList, unsigned char * buf, unsigned int sz, int flag, int max)This function is used to set aside static memory for wolfCrypt use with custom bucket sizes and distributions. Memory can be used by passing the created heap hint into functions. This extended version allows for custom bucket sizes and distributions instead of using the default predefined sizes.
WOLFSSL_HEAP_HINT *	**wolfSSL_SetGlobalHeapHint**(WOLFSSL_HEAP_HINT * hint)This function sets a global heap hint that will be used when NULL heap hint is passed to memory allocation functions. This allows for setting a default heap hint that will be used across the entire application.
WOLFSSL_HEAP_HINT *	**wolfSSL_GetGlobalHeapHint**(void)This function gets the current global heap hint that is used when NULL heap hint is passed to memory allocation functions.
void	**wolfSSL_SetDebugMemoryCb**(DebugMemoryCb cb)This function sets a debug callback function for static memory allocation tracking. Used with WOLFSSL_STATIC_MEMORY_DEBUG_CALLBACK build option. The callback function will be called during memory allocation and deallocation operations to provide debugging information.

	Name
void	**wc_UnloadStaticMemory**(WOLFSSL_HEAP_HINT * heap)This function frees static memory heap and associated mutex. Should be called when done using static memory allocation to properly clean up resources.
int	**wolfSSL_StaticBufferSz_ex**(unsigned int listSz, const word32 * sizeList, const word32 * distList, byte * buffer, word32 sz, int flag)This function calculates the required buffer size for static memory allocation with custom bucket sizes and distributions. This extended version allows for custom bucket sizes instead of using the default predefined sizes.
void *	**XMALLOC**(size_t n, void * heap, int type)This is not actually a function, but rather a preprocessor macro, which allows the user to substitute in their own malloc, realloc, and free functions in place of the standard C memory functions. To use external memory functions, define XMALLOC_USER. This will cause the memory functions to be replaced by external functions of the form: extern void *XMALLOC(size_t n, void* heap, int type); extern void _XREALLOC(void *p, size_t n, void_ heap, int type); extern void XFREE(void* p, void *heap, int type); To use the basic C memory functions in place of wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free, define NO_WOLFSSL_MEMORY. This will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) free((xp));} #define XREALLOC(p, n, h, t) realloc((p), (n)) If none of these options are selected, the system will default to use the wolfSSL memory functions. A user can set custom memory functions through callback hooks, (see wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free). This option will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, wolfSSL_Malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) wolfSSL_Free((xp));} #define XREALLOC(p, n, h, t) wolfSSL_Realloc((p), (n))

	Name
void *	**XREALLOC**(void * p, size_t n, void * heap, int type)This is not actually a function, but rather a preprocessor macro, which allows the user to substitute in their own malloc, realloc, and free functions in place of the standard C memory functions. To use external memory functions, define XMALLOC_USER. This will cause the memory functions to be replaced by external functions of the form: extern void *XMALLOC(size_t n, void* heap, int type); extern void _XREALLOC(void *p, size_t n, void_ heap, int type); extern void XFREE(void* p, void *heap, int type); To use the basic C memory functions in place of wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free, define NO_WOLFSSL_MEMORY. This will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) free((xp));} #define XREALLOC(p, n, h, t) realloc((p), (n)) If none of these options are selected, the system will default to use the wolfSSL memory functions. A user can set custom memory functions through callback hooks, (see wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free). This option will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, wolfSSL_Malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) wolfSSL_Free((xp));} #define XREALLOC(p, n, h, t) wolfSSL_Realloc((p), (n))

	Name
void	**XFREE**(void * p, void * heap, int type)This is not actually a function, but rather a preprocessor macro, which allows the user to substitute in their own malloc, realloc, and free functions in place of the standard C memory functions. To use external memory functions, define XMALLOC_USER. This will cause the memory functions to be replaced by external functions of the form: extern void *XMALLOC(size_t n, void* heap, int type); extern void _XREALLOC(void *p, size_t n, void_ heap, int type); extern void XFREE(void* p, void *heap, int type); To use the basic C memory functions in place of wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free, define NO_WOLFSSL_MEMORY. This will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) free((xp));} #define XREALLOC(p, n, h, t) realloc((p), (n)) If none of these options are selected, the system will default to use the wolfSSL memory functions. A user can set custom memory functions through callback hooks, (see wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free). This option will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, wolfSSL_Malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) wolfSSL_Free((xp));} #define XREALLOC(p, n, h, t) wolfSSL_Realloc((p), (n))
long	**wolfCrypt_heap_peakAllocs_checkpoint**(void)Checkpoints peak heap allocations.
long	**wolfCrypt_heap_peakBytes_checkpoint**(void)Checkpoints peak heap bytes.

A.2.2 Functions Documentation

```
void * wolfSSL_Malloc(
    size_t size,
    void * heap,
    int type
)
```

This function is similar to malloc(), but calls the memory allocation function which wolfSSL has been configured to use. By default, wolfSSL uses malloc(). This can be changed using the wolfSSL memory abstraction layer - see wolfSSL_SetAllocators(). Note wolfSSL_Malloc is not called directly by wolfSSL, but instead called by macro XMALLOC. For the default build only the size argument exists. If using WOLFSSL_STATIC_MEMORY build then heap and type arguments are included.

Parameters:

- **size** size, in bytes, of the memory to allocate

- **heap** heap hint to use for memory. Can be NULL
- **type** dynamic type (see DYNAMIC_TYPE_ list in types.h)

See:

- wolfSSL_Free
- wolfSSL_Realloc
- wolfSSL_SetAllocators
- XMALLOC
- XFREE
- XREALLOC

Return:

- pointer If successful, this function returns a pointer to allocated memory.
- error If there is an error, NULL will be returned.

Example

```
int* tenInts = (int*)wolfSSL_Malloc(sizeof(int)*10);
```

```
void wolfSSL_Free(
    void * ptr,
    void * heap,
    int type
)
```

This function is similar to free(), but calls the memory free function which wolfSSL has been configured to use. By default, wolfSSL uses free(). This can be changed using the wolfSSL memory abstraction layer - see wolfSSL_SetAllocators(). Note wolfSSL_Free is not called directly by wolfSSL, but instead called by macro XFREE. For the default build only the ptr argument exists. If using WOLFSSL_STATIC_MEMORY build then heap and type arguments are included.

Parameters:

- **ptr** pointer to the memory to be freed.
- **heap** heap hint to use for memory. Can be NULL
- **type** dynamic type (see DYNAMIC_TYPE_ list in types.h)

See:

- wolfSSL_Alloc
- wolfSSL_Realloc
- wolfSSL_SetAllocators
- XMALLOC
- XFREE
- XREALLOC

Return: none No returns.

Example

```
int* tenInts = (int*)wolfSSL_Malloc(sizeof(int)*10);
// process data as desired
...
if(tenInts) {
    wolfSSL_Free(tenInts, NULL, DYNAMIC_TYPE_TMP_BUFFER);
}
```

```
void * wolfSSL_Realloc(
    void * ptr,
    size_t size,
    void * heap,
    int type
)
```

This function is similar to realloc(), but calls the memory re-allocation function which wolfSSL has been configured to use. By default, wolfSSL uses realloc(). This can be changed using the wolfSSL memory abstraction layer - see wolfSSL_SetAllocators(). Note wolfSSL_Realloc is not called directly by wolfSSL, but instead called by macro XREALLOC. For the default build only the size argument exists. If using WOLFSSL_STATIC_MEMORY build then heap and type arguments are included.

Parameters:

- **ptr** pointer to the previously-allocated memory, to be reallocated.
- **size** number of bytes to allocate.
- **heap** heap hint to use for memory. Can be NULL
- **type** dynamic type (see DYNAMIC_TYPE_ list in types.h)

See:

- wolfSSL_Free
- wolfSSL_Malloc
- wolfSSL_SetAllocators
- XMALLOC
- XFREE
- XREALLOC

Return:

- pointer If successful, this function returns a pointer to re-allocated memory. This may be the same pointer as ptr, or a new pointer location.
- Null If there is an error, NULL will be returned.

Example

```
int* tenInts = (int*)wolfSSL_Malloc(sizeof(int)*10);
int* twentyInts = (int*)wolfSSL_Realloc(tenInts, sizeof(int)*20);
```

```
int wolfSSL_SetAllocators(
    wolfSSL_Malloc_cb mf,
    wolfSSL_Free_cb ff,
    wolfSSL_Realloc_cb rf
)
```

This function registers the allocation functions used by wolfSSL. By default, if the system supports it, malloc/free and realloc are used. Using this function allows the user at runtime to install their own memory handlers.

Parameters:

- **malloc_function** memory allocation function for wolfSSL to use. Function signature must match wolfSSL_Malloc_cb prototype, above.
- **free_function** memory free function for wolfSSL to use. Function signature must match wolfSSL_Free_cb prototype, above.
- **realloc_function** memory re-allocation function for wolfSSL to use. Function signature must match wolfSSL_Realloc_cb prototype, above.

See: none

Return:

- Success If successful this function will return 0.
- BAD_FUNC_ARG is the error that will be returned if a function pointer is not provided.

Example

```
static void* MyMalloc(size_t size)
{
    // custom malloc function
}

static void MyFree(void* ptr)
{
    // custom free function
}

static void* MyRealloc(void* ptr, size_t size)
{
    // custom realloc function
}

// Register custom memory functions with wolfSSL
int ret = wolfSSL_SetAllocators(MyMalloc, MyFree, MyRealloc);
if (ret != 0) {
    // failed to set memory functions
}
```

```
int wolfSSL_StaticBufferSz(
    byte * buffer,
```

```
    word32 sz,
    int flag
)
```

This function is available when static memory feature is used (–enable-staticmemory). It gives the optimum buffer size for memory “buckets”. This allows for a way to compute buffer size so that no extra unused memory is left at the end after it has been partitioned. For the none _ex version of this function the default bucket and distribution list set during compile time is used. The returned value, if positive, is the computed buffer size to use.

Parameters:

- **buffer** pointer to buffer
- **size** size of buffer
- **type** desired type of memory ie WOLFMEM_GENERAL or WOLFMEM_IO_POOL

See:

- wolfSSL_Malloc
- wolfSSL_Free

Return:

- Success On successfully completing buffer size calculations a positive value is returned. This returned value is for optimum buffer size.
- Failure All negative values are considered to be error cases.

Example

```
byte buffer[1000];
word32 size = sizeof(buffer);
int optimum;

optimum = wolfSSL_StaticBufferSz(buffer, size, WOLFMEM_GENERAL);
if (optimum < 0) { //handle error case }
printf(“The optimum buffer size to make use of all memory is %d\n”,
optimum);
...
```

```
int wolfSSL_MemoryPaddingSz(
    void
)
```

This function is available when static memory feature is used (–enable-staticmemory). It gives the size of padding needed for each partition of memory. This padding size will be the size needed to contain a memory management structure along with any extra for memory alignment.

Parameters:

- **none** No parameters.

See:

- wolfSSL_Malloc
- wolfSSL_Free

Return:

- On successfully memory padding calculation the return value will be a positive value
- All negative values are considered error cases.

Example

```
int padding;
padding = wolfSSL_MemoryPaddingSz();
if (padding < 0) { //handle error case }
printf("The padding size needed for each \"bucket\" of memory is %d\n",
padding);
// calculation of buffer for IO POOL size is number of buckets
// times (padding + WOLFMEM_IO_SZ)
...
```

```
int wolfSSL_CTX_load_static_memory(
    WOLFSSL_CTX ** ctx,
    wolfSSL_method_func method,
    unsigned char * buf,
    unsigned int sz,
    int flag,
    int max
)
```

This function is used to set aside static memory for a CTX. Memory set aside is then used for the CTX's lifetime and for any SSL objects created from the CTX. By passing in a NULL ctx pointer and a wolfSSL_method_func function the creation of the CTX itself will also use static memory. wolfSSL_method_func has the function signature of WOLFSSL_METHOD* (*wolfSSL_method_func)(void* heap);. Passing in 0 for max makes it behave as if not set and no max concurrent use restrictions is in place. The flag value passed in determines how the memory is used and behavior while operating. Available flags are the following.

Parameters:

- **ctx** address of pointer to a WOLFSSL_CTX structure.
- **method** function to create protocol. (should be NULL if ctx is not also NULL)
- **buf** memory to use for all operations.
- **sz** size of memory buffer being passed in.
- **flag** type of memory.
- **max** max concurrent operations.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_is_static_memory
- wolfSSL_is_static_memory

Return:

- If successful, SSL_SUCCESS will be returned.
- All unsuccessful return values will be less than 0 or equal to SSL_FAILURE.

0 - default general memory

WOLFMEM_IO_POOL - used for input/output buffer when sending receiving messages. Overrides general memory, so all memory in buffer passed in is used for IO. WOLFMEM_IO_FIXED - same as WOLFMEM_IO_POOL but each SSL now keeps two buffers to themselves for their lifetime. WOLFMEM_TRACK_STATS - each SSL keeps track of memory stats while running.

Example

```
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
int ret;
unsigned char memory[MAX];
int memorySz = MAX;
unsigned char IO[MAX];
int IOSz = MAX;
int flag = WOLFMEM_IO_FIXED | WOLFMEM_TRACK_STATS;
...
// create ctx also using static memory, start with general memory to use
ctx = NULL:
ret = wolfSSL_CTX_load_static_memory(&ctx, wolfSSLv23_server_method_ex,
 ↪  memory, memorySz, 0,
MAX_CONCURRENT_HANDSHAKES);
if (ret != SSL_SUCCESS) {
// handle error case
}
// load in memory for use with IO
ret = wolfSSL_CTX_load_static_memory(&ctx, NULL, IO, IOSz, flag,
 ↪  MAX_CONCURRENT_IO);
if (ret != SSL_SUCCESS) {
// handle error case
}
...
```

```
int wolfSSL_CTX_is_static_memory(
    WOLFSSL_CTX * ctx,
    WOLFSSL_MEM_STATS * mem_stats
)
```

This function does not change any of the connections behavior and is used only for gathering information about the static memory usage.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **mem_stats** structure to hold information about staic memory usage.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_load_static_memory
- wolfSSL_is_static_memory

Return:

- A value of 1 is returned if using static memory for the CTX is true.
- 0 is returned if not using static memory.

Example

```
WOLFSSL_CTX* ctx;
int ret;
WOLFSSL_MEM_STATS mem_stats;
...
//get information about static memory with CTX

ret = wolfSSL_CTX_is_static_memory(ctx, &mem_stats);

if (ret == 1) {
    // handle case of is using static memory
    // print out or inspect elements of mem_stats
}

if (ret == 0) {
    //handle case of ctx not using static memory
}
...
```

```
int wolfSSL_is_static_memory(
    WOLFSSL * ssl,
    WOLFSSL_MEM_CONN_STATS * mem_stats
)
```

wolfSSL_is_static_memory is used to gather information about a SSL's static memory usage. The return value indicates if static memory is being used and WOLFSSL_MEM_CONN_STATS will be filled out if and only if the flag WOLFMEM_TRACK_STATS was passed to the parent CTX when loading in static memory.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **mem_stats** structure to contain static memory usage

See:

- wolfSSL_new
- wolfSSL_CTX_is_static_memory

Return:

- A value of 1 is returned if using static memory for the CTX is true.
- 0 is returned if not using static memory.

Example

```
WOLFSSL* ssl;
int ret;
WOLFSSL_MEM_CONN_STATS mem_stats;

...

ret = wolfSSL_is_static_memory(ssl, mem_stats);

if (ret == 1) {
    // handle case when is static memory
    // investigate elements in mem_stats if WOLFMEM_TRACK_STATS flag
}
...
```

```
int wc_LoadStaticMemory(
    WOLFSSL_HEAP_HINT ** pHint,
    unsigned char * buf,
    unsigned int sz,
    int flag,
    int max
)
```

This function is used to set aside static memory for wolfCrypt use. Memory can be used by passing the created heap hint into functions. An example of this is when calling wc_InitRng_ex. The flag value passed in determines how the memory is used and behavior while operating, in general wolfCrypt operations will use memory from a WOLFMEM_GENERAL pool. Available flags are the following.

Parameters:

- **pHint** WOLFSSL_HEAP_HINT structure to use
- **buf** memory to use for all operations.
- **sz** size of memory buffer being passed in.
- **flag** type of memory.
- **max** max concurrent operations (handshakes, IO).

See: none

Return:

- Returns 0 on success.
- Returns a non-zero integer on failure.

WOLFMEM_GENERAL - default general memory

WOLFMEM_IO_POOL - used for input/output buffer when sending receiving messages. Overrides general memory, so all memory in buffer passed in is used for IO. WOLFMEM_IO_FIXED - same as WOLFMEM_IO_POOL but each SSL now keeps two buffers to themselves for their lifetime. WOLFMEM_TRACK_STATS - each SSL keeps track of memory stats while running

Example

```
WOLFSSL_HEAP_HINT hint;
int ret;
unsigned char memory[MAX];
int memorySz = MAX;
int flag = WOLFMEM_GENERAL | WOLFMEM_TRACK_STATS;
...

// load in memory for use

ret = wc_LoadStaticMemory(&hint, memory, memorySz, flag, 0);
if (ret) {
    // handle error case
}
...

ret = wc_InitRng_ex(&rng, hint, 0);

// check ret value
```

```
int wc_LoadStaticMemory_ex(
    WOLFSSL_HEAP_HINT ** pHint,
    unsigned int listSz,
    const word32 * sizeList,
    const word32 * distList,
    unsigned char * buf,
    unsigned int sz,
    int flag,
    int max
)
```

This function is used to set aside static memory for wolfCrypt use with custom bucket sizes and distributions. Memory can be used by passing the created heap hint into functions. This extended version allows for custom bucket sizes and distributions instead of using the default predefined sizes.

Parameters:

- **pHint** WOLFSSL_HEAP_HINT handle to initialize
- **listSz** number of entries in the size and distribution lists
- **sizeList** array of bucket sizes to use

- **distList** distribution list matching sizeList
- **buf** memory to use for all operations.
- **sz** size of memory buffer being passed in.
- **flag** type of memory.
- **max** max concurrent operations (handshakes, IO).

See:

- wc_LoadStaticMemory
- wc_UnloadStaticMemory

Return:

- Returns 0 on success.
- Returns a non-zero integer on failure.

Example

```
WOLFSSL_HEAP_HINT hint;
int ret;
unsigned char memory[MAX];
int memorySz = MAX;
int flag = WOLFMEM_GENERAL | WOLFMEM_TRACK_STATS;
const word32 sizeList[] = {64, 128, 256, 512, 1024};
const word32 distList[] = {1, 1, 1, 1, 1};
unsigned int listSz = (unsigned int)(sizeof(sizeList)/
                                     sizeof(sizeList[0]));
...

// load in memory for use with custom bucket sizes

ret = wc_LoadStaticMemory_ex(&hint, listSz, sizeList, distList,
                             memory, memorySz, flag, 0);
if (ret) {
    // handle error case
}
...

ret = wc_InitRng_ex(&rng, hint, 0);

// check ret value
```

```
WOLFSSL_HEAP_HINT * wolfSSL_SetGlobalHeapHint(
    WOLFSSL_HEAP_HINT * hint
)
```

This function sets a global heap hint that will be used when NULL heap hint is passed to memory allocation functions. This allows for setting a default heap hint that will be used across the entire application.

Parameters:

- **hint** WOLFSSL_HEAP_HINT structure to use as the global heap hint

See:

- wolfSSL_GetGlobalHeapHint
- wc_LoadStaticMemory

Return: Returns the previous global heap hint that was set.

Example

```
WOLFSSL_HEAP_HINT hint;
WOLFSSL_HEAP_HINT* prev_hint;
int ret;
unsigned char memory[MAX];
int memorySz = MAX;
...

// load in memory for use
ret = wc_LoadStaticMemory(&hint, memory, memorySz, WOLFMEM_GENERAL, 0);
if (ret != SSL_SUCCESS) {
    // handle error case
}

// set as global heap hint
prev_hint = wolfSSL_SetGlobalHeapHint(&hint);
if (prev_hint != NULL) {
    // there was a previous global heap hint
}
```

```
WOLFSSL_HEAP_HINT * wolfSSL_GetGlobalHeapHint(
    void
)
```

This function gets the current global heap hint that is used when NULL heap hint is passed to memory allocation functions.

Parameters:

- **none** No parameters.

See:

- wolfSSL_SetGlobalHeapHint
- wc_LoadStaticMemory

Return: Returns the current global heap hint, or NULL if none is set.

Example

```
WOLFSSL_HEAP_HINT* current_hint;
...

current_hint = wolfSSL_GetGlobalHeapHint();
if (current_hint != NULL) {
    // there is a global heap hint set
    // can use current_hint for operations
}
```

```
void wolfSSL_SetDebugMemoryCb(
    DebugMemoryCb cb
)
```

This function sets a debug callback function for static memory allocation tracking. Used with WOLFSSL_STATIC_MEMORY_DEBUG_CALLBACK build option. The callback function will be called during memory allocation and deallocation operations to provide debugging information.

Parameters:

- **cb** debug callback function to set

See: none

Return:

- If successful, 0 will be returned.
- All unsuccessful return values will be less than 0.

Example

```
static void debug_memory_cb(const char* func, const char* file, int line,
                            void* ptr, size_t size, int type)
{
    printf("Memory %s: %s:%d ptr=%p size=%zu type=%d\n",
           func, file, line, ptr, size, type);
}
...

// set debug callback
int ret = wolfSSL_SetDebugMemoryCb(debug_memory_cb);
if (ret != 0) {
    // handle error case
}
```

```
void wc_UnloadStaticMemory(
    WOLFSSL_HEAP_HINT * heap
)
```

This function frees static memory heap and associated mutex. Should be called when done using static memory allocation to properly clean up resources.

Parameters:

- **hint** WOLFSSL_HEAP_HINT structure to unload

See:

- wc_LoadStaticMemory
- wc_LoadStaticMemory_ex

Return:

- If successful, 0 will be returned.
- All unsuccessful return values will be less than 0.

Example

```
WOLFSSL_HEAP_HINT hint;
int ret;
unsigned char memory[MAX];
int memorySz = MAX;
...

// load in memory for use
ret = wc_LoadStaticMemory(&hint, memory, memorySz, WOLFMEM_GENERAL, 0);
if (ret != SSL_SUCCESS) {
    // handle error case
}

// use memory for operations
...

// cleanup when done
wc_UnloadStaticMemory(&hint);
```

```
int wolfSSL_StaticBufferSz_ex(
    unsigned int listSz,
    const word32 * sizeList,
    const word32 * distList,
    byte * buffer,
    word32 sz,
    int flag
)
```

This function calculates the required buffer size for static memory allocation with custom bucket sizes and distributions. This extended version allows for custom bucket sizes instead of using the default predefined sizes.

Parameters:

- **bucket_sizes** array of bucket sizes to use
- **bucket_count** number of bucket sizes in the array
- **flag** desired type of memory ie WOLFMEM_GENERAL or WOLFMEM_IO_POOL

See:

- wolfSSL_StaticBufferSz
- wc_LoadStaticMemory_ex

Return:

- On successfully completing buffer size calculations a positive value is returned.
- All negative values are considered to be error cases.

Example

```
word32 sizeList[] = {64, 128, 256, 512, 1024};
word32 distList[] = {1, 2, 1, 1, 1};
int listSz = 5;
int optimum;

optimum = wolfSSL_StaticBufferSz_ex(listSz, sizeList, distList, NULL, 0,
    WOLFMEM_GENERAL);
if (optimum < 0) { //handle error case }
printf("The optimum buffer size with custom buckets is %d\n", optimum);
...
```

```
void * XMALLOC(
    size_t n,
    void * heap,
    int type
)
```

This is not actually a function, but rather a preprocessor macro, which allows the user to substitute in their own malloc, realloc, and free functions in place of the standard C memory functions. To use external memory functions, define XMALLOC_USER. This will cause the memory functions to be replaced by external functions of the form: extern void *XMALLOC(size_t n, void* heap, int type); extern void _XREALLOC(void *p, size_t n, void_ heap, int type); extern void XFREE(void* p, void *heap, int type); To use the basic C memory functions in place of wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free, define NO_WOLFSSL_MEMORY. This will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) free((xp));} #define XREALLOC(p, n, h, t) realloc((p), (n)) If none of these options are selected, the system will default to use the wolfSSL memory functions. A user can set custom memory functions through callback hooks, (see wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free). This option will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, wolfSSL_Malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) wolfSSL_Free((xp));} #define XREALLOC(p, n, h, t) wolfSSL_Realloc((p), (n))

Parameters:

- **s** size of memory to allocate
- **h** (used by custom XMALLOC function) pointer to the heap to use
- **t** memory allocation types for user hints. See enum in types.h

See:

- wolfSSL_Malloc
- wolfSSL_Realloc
- wolfSSL_Free
- wolfSSL_SetAllocators

Return:

- pointer Return a pointer to allocated memory on success
- NULL on failure

Example

```
int* tenInts = XMALLOC(sizeof(int)*10, NULL, DYNAMIC_TYPE_TMP_BUFFER);
if (tenInts == NULL) {
    // error allocating space
    return MEMORY_E;
}
```

```
void * XREALLOC(
    void * p,
    size_t n,
    void * heap,
    int type
)
```

This is not actually a function, but rather a preprocessor macro, which allows the user to substitute in their own malloc, realloc, and free functions in place of the standard C memory functions. To use external memory functions, define XMALLOC_USER. This will cause the memory functions to be replaced by external functions of the form: extern void *XMALLOC(size_t n, void* heap, int type); extern void _XREALLOC(void *p, size_t n, void_ heap, int type); extern void XFREE(void* p, void *heap, int type); To use the basic C memory functions in place of wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free, define NO_WOLFSSL_MEMORY. This will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) free((xp));} #define XREALLOC(p, n, h, t) realloc((p), (n)) If none of these options are selected, the system will default to use the wolfSSL memory functions. A user can set custom memory functions through callback hooks, (see wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free). This option will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, wolfSSL_Malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) wolfSSL_Free((xp));} #define XREALLOC(p, n, h, t) wolfSSL_Realloc((p), (n))

Parameters:

- **p** pointer to the address to reallocate
- **n** size of memory to allocate
- **h** (used by custom XREALLOC function) pointer to the heap to use

- **t** memory allocation types for user hints. See enum in types.h

See:

- wolfSSL_Malloc
- wolfSSL_Realloc
- wolfSSL_Free
- wolfSSL_SetAllocators

Return:

- Return a pointer to allocated memory on success
- NULL on failure

Example

```
int* tenInts = (int*)XMALLOC(sizeof(int)*10, NULL, DYNAMIC_TYPE_TMP_BUFFER);
int* twentyInts = (int*)XREALLOC(tenInts, sizeof(int)*20, NULL,
    DYNAMIC_TYPE_TMP_BUFFER);
```

```
void XFREE(
    void * p,
    void * heap,
    int type
)
```

This is not actually a function, but rather a preprocessor macro, which allows the user to substitute in their own malloc, realloc, and free functions in place of the standard C memory functions. To use external memory functions, define XMALLOC_USER. This will cause the memory functions to be replaced by external functions of the form: extern void *XMALLOC(size_t n, void* heap, int type); extern void _XREALLOC(void *p, size_t n, void_ heap, int type); extern void XFREE(void* p, void *heap, int type); To use the basic C memory functions in place of wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free, define NO_WOLFSSL_MEMORY. This will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) free((xp));} #define XREALLOC(p, n, h, t) realloc((p), (n)) If none of these options are selected, the system will default to use the wolfSSL memory functions. A user can set custom memory functions through callback hooks, (see wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free). This option will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, wolfSSL_Malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) wolfSSL_Free((xp));} #define XREALLOC(p, n, h, t) wolfSSL_Realloc((p), (n))

Parameters:

- **p** pointer to the address to free
- **h** (used by custom XFREE function) pointer to the heap to use
- **t** memory allocation types for user hints. See enum in types.h

See:

- wolfSSL_Malloc
- wolfSSL_Realloc

- wolfSSL_Free
- wolfSSL_SetAllocators

Return: none No returns.

Example

```
int* tenInts = XMALLOC(sizeof(int) * 10, NULL, DYNAMIC_TYPE_TMP_BUFFER);
if (tenInts == NULL) {
    // error allocating space
    return MEMORY_E;
}
```

```
long wolfCrypt_heap_peakAllocs_checkpoint(
    void
)
```

Checkpoints peak heap allocations.

Parameters:

- **none** No parameters

See: wolfCrypt_heap_peakBytes_checkpoint

Return: Peak allocation count

Example

```
long peak = wolfCrypt_heap_peakAllocs_checkpoint();
```

```
long wolfCrypt_heap_peakBytes_checkpoint(
    void
)
```

Checkpoints peak heap bytes.

Parameters:

- **none** No parameters

See: wolfCrypt_heap_peakAllocs_checkpoint

Return: Peak bytes allocated

Example

```
long peak = wolfCrypt_heap_peakBytes_checkpoint();
```

A.3 OpenSSL API

A.2.2.21 function wolfCrypt_heap_peakBytes_checkpoint

A.3.1 Functions

	Name
int	**wolfSSL_BN_mod_exp**(WOLFSSL_BIGNUM * r, const WOLFSSL_BIGNUM * a, const WOLFSSL_BIGNUM * p, const WOLFSSL_BIGNUM * m, WOLFSSL_BN_CTX * ctx)This function performs the following math "r = (a^p) % m".
const WOLFSSL_EVP_CIPHER *	**wolfSSL_EVP_des_ede3_ecb**(void)Getter functions for the respective WOLFSSL_EVP_CIPHER pointers. wolfSSL_EVP_init() must be called once in the program first to populate these cipher strings. WOLFSSL_DES_ECB macro must be defined for wolfSSL_EVP_des_ede3_ecb().
const WOLFSSL_EVP_CIPHER *	**wolfSSL_EVP_des_cbc**(void)Getter functions for the respective WOLFSSL_EVP_CIPHER pointers. wolfSSL_EVP_init() must be called once in the program first to populate these cipher strings. WOLFSSL_DES_ECB macro must be defined for wolfSSL_EVP_des_ecb().
int	**wolfSSL_EVP_DigestInit_ex**(WOLFSSL_EVP_MD_CTX * ctx, const WOLFSSL_EVP_MD * type, WOLFSSL_ENGINE * impl)Function for initializing WOLFSSL_EVP_MD_CTX. This function is a wrapper for wolfSSL_EVP_DigestInit() because wolfSSL does not use WOLFSSL_ENGINE.
int	**wolfSSL_EVP_CipherInit_ex**(WOLFSSL_EVP_CIPHER_CTX * ctx, const WOLFSSL_EVP_CIPHER * type, WOLFSSL_ENGINE * impl, const unsigned char * key, const unsigned char * iv, int enc)Function for initializing WOLFSSL_EVP_CIPHER_CTX. This function is a wrapper for wolfSSL_CipherInit() because wolfSSL does not use WOLFSSL_ENGINE.
int	**wolfSSL_EVP_EncryptInit_ex**(WOLFSSL_EVP_CIPHER_CTX * ctx, const WOLFSSL_EVP_CIPHER * type, WOLFSSL_ENGINE * impl, const unsigned char * key, const unsigned char * iv)Function for initializing WOLFSSL_EVP_CIPHER_CTX. This function is a wrapper for wolfSSL_EVP_CipherInit() because wolfSSL does not use WOLFSSL_ENGINE. Sets encrypt flag to be encrypt.

	Name
int	**wolfSSL_EVP_DecryptInit_ex**(WOLFSSL_EVP_CIPHER_CTX * ctx, const WOLFSSL_EVP_CIPHER * type, WOLFSSL_ENGINE * impl, const unsigned char * key, const unsigned char * iv)Function for initializing WOLFSSL_EVP_CIPHER_CTX. This function is a wrapper for wolfSSL_EVP_CipherInit() because wolfSSL does not use WOLFSSL_ENGINE. Sets encrypt flag to be decrypt.
int	**wolfSSL_EVP_CipherUpdate**(WOLFSSL_EVP_CIPHER_CTX * ctx, unsigned char * out, int * outl, const unsigned char * in, int inl)Function for encrypting/decrypting data. In buffer is added to be encrypted or decrypted and out buffer holds the results. outl will be the length of encrypted/decrypted information.
int	**wolfSSL_EVP_CipherFinal**(WOLFSSL_EVP_CIPHER_CTX * ctx, unsigned char * out, int * outl)This function performs the final cipher operations adding in padding. If WOLFSSL_EVP_CIPH_NO_PADDING flag is set in WOLFSSL_EVP_CIPHER_CTX structure then 1 is returned and no encryption/decryption is done. If padding flag is seti padding is added and encrypted when ctx is set to encrypt, padding values are checked when set to decrypt.
int	**wolfSSL_EVP_CIPHER_CTX_set_key_length**(WOLFSSL_EVP_CIPHE * ctx, int keylen)Setter function for WOLFSSL_EVP_CIPHER_CTX structure key length.
int	**wolfSSL_EVP_CIPHER_CTX_block_size**(const WOLFSSL_EVP_CIPHER_CTX * ctx)This is a getter function for the ctx block size.
int	**wolfSSL_EVP_CIPHER_block_size**(const WOLFSSL_EVP_CIPHER * cipher)This is a getter function for the block size of cipher.
void	**wolfSSL_EVP_CIPHER_CTX_set_flags**(WOLFSSL_EVP_CIPHER_CTX * ctx, int flags)Setter function for WOLFSSL_EVP_CIPHER_CTX structure.
void	**wolfSSL_EVP_CIPHER_CTX_clear_flags**(WOLFSSL_EVP_CIPHER_C * ctx, int flags)Clearing function for WOLFSSL_EVP_CIPHER_CTX structure.
int	**wolfSSL_EVP_CIPHER_CTX_set_padding**(WOLFSSL_EVP_CIPHER_ * c, int pad)Setter function for WOLFSSL_EVP_CIPHER_CTX structure to use padding.
unsigned long	**wolfSSL_EVP_CIPHER_CTX_flags**(const WOLFSSL_EVP_CIPHER_CTX * ctx)Getter function for WOLFSSL_EVP_CIPHER_CTX structure. Deprecated v1.1.0.

	Name
int	**wolfSSL_PEM_write_bio_PrivateKey**(WOLFSSL_BIO * bio, WOLFSSL_EVP_PKEY * key, const WOLFSSL_EVP_CIPHER * cipher, unsigned char * passwd, int len, wc_pem_password_cb * cb, void * arg)This function writes a key into a WOLFSSL_BIO structure in PEM format.
int	**wolfSSL_CTX_use_RSAPrivateKey_file**(WOLFSSL_CTX * ctx, const char * file, int format)This function loads the private RSA key used in the SSL connection into the SSL context (WOLFSSL_CTX). This function is only available when wolfSSL has been compiled with the OpenSSL compatibility layer enabled (–enable_opensslExtra, #define OPENSSL_EXTRA), and is identical to the more_typically used wolfSSL_CTX_use_PrivateKey_file() function. The file argument contains a pointer to the RSA private key file, in the format specified by format.
int	**wolfSSL_use_certificate_file**(WOLFSSL * ssl, const char * file, int format)This function loads a certificate file into the SSL session (WOLFSSL structure). The certificate file is provided by the file argument. The format argument specifies the format type of the file - either SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.
int	**wolfSSL_use_PrivateKey_file**(WOLFSSL * ssl, const char * file, int format)This function loads a private key file into the SSL session (WOLFSSL structure). The key file is provided by the file argument. The format argument specifies the format type of the file - SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.
int	**wolfSSL_use_certificate_chain_file**(WOLFSSL * ssl, const char * file)This function loads a chain of certificates into the SSL session (WOLFSSL structure). The file containing the certificate chain is provided by the file argument, and must contain PEM_formatted certificates. This function will process up to MAX_CHAIN_DEPTH (default = 9, defined in internal.h) certificates, plus the subject certificate.

	Name
int	**wolfSSL_use_RSAPrivateKey_file**(WOLFSSL * ssl, const char * file, int format)This function loads the private RSA key used in the SSL connection into the SSL session (WOLFSSL structure). This function is only available when wolfSSL has been compiled with the OpenSSL compatibility layer enabled (–enable_opensslExtra, #define OPENSSL_EXTRA), and is identical to the more_typically used wolfSSL_use_PrivateKey_file() function. The file argument contains a pointer to the RSA private key file, in the format specified by format.
long	**wolfSSL_set_tlsext_status_type**(WOLFSSL * s, int type)This function is called when the client application request that a server send back an OCSP status response (also known as OCSP stapling).Currently, the only supported type is TLSEXT_STATUSTYPE_ocsp.
WOLFSSL_X509_CHAIN *	**wolfSSL_get_peer_chain**(WOLFSSL * ssl)Retrieves the peer's certificate chain.
int	**wolfSSL_get_chain_count**(WOLFSSL_X509_CHAIN * chain)Retrieve's the peers certificate chain count.
int	**wolfSSL_get_chain_length**(WOLFSSL_X509_CHAIN * chain, int idx)Retrieves the peer's ASN1.DER certificate length in bytes at index (idx).
unsigned char *	**wolfSSL_get_chain_cert**(WOLFSSL_X509_CHAIN * chain, int idx)Retrieves the peer's ASN1.DER certificate at index (idx).
int	**wolfSSL_get_chain_cert_pem**(WOLFSSL_X509_CHAIN * chain, int idx, unsigned char * buf, int inLen, int * outLen)Retrieves the peer's PEM certificate at index (idx).
const unsigned char *	**wolfSSL_get_sessionID**(const WOLFSSL_SESSION * s)Retrieves the session's ID. The session ID is always 32 bytes long.
int	**wolfSSL_X509_get_serial_number**(WOLFSSL_X509 * x509, unsigned char * in, int * inOutSz)Retrieves the peer's certificate serial number. The serial number buffer (in) should be at least 32 bytes long and be provided as the *inOutSz argument as input. After calling the function* inOutSz will hold the actual length in bytes written to the in buffer.

	Name
WC_PKCS12 *	**wolfSSL_d2i_PKCS12_bio**(WOLFSSL_BIO * bio, WC_PKCS12 ** pkcs12)wolfSSL_d2i_PKCS12_bio (d2i_PKCS12_bio) copies in the PKCS12 information from WOLFSSL_BIO to the structure WC_PKCS12. The information is divided up in the structure as a list of Content Infos along with a structure to hold optional MAC information. After the information has been divided into chunks (but not decrypted) in the structure WC_PKCS12, it can then be parsed and decrypted by calling.
WC_PKCS12 *	**wolfSSL_i2d_PKCS12_bio**(WOLFSSL_BIO * bio, WC_PKCS12 * pkcs12)wolfSSL_i2d_PKCS12_bio (i2d_PKCS12_bio) copies in the cert information from the structure WC_PKCS12 to WOLFSSL_BIO.
int	**wolfSSL_PKCS12_parse**(WC_PKCS12 * pkcs12, const char * psw, WOLFSSL_EVP_PKEY ** pkey, WOLFSSL_X509 ** cert, WOLF_STACK_OF(WOLFSSL_X509) ** ca)PKCS12 can be enabled with adding –enable_opensslextra to the configure command. It can use triple DES and RC4 for decryption so would recommend also enabling these features when enabling opensslextra (–enable_des3 –enable_arc4). wolfSSL does not currently support RC2 so decryption with RC2 is currently not available. This may be noticeable with default encryption schemes used by OpenSSL command line to create .p12 files. wolfSSL_PKCS12_parse (PKCS12_parse). The first thing this function does is check the MAC is correct if present. If the MAC fails then the function returns and does not try to decrypt any of the stored Content Infos. This function then parses through each Content Info looking for a bag type, if the bag type is known it is decrypted as needed and either stored in the list of certificates being built or as a key found. After parsing through all bags the key found is then compared with the certificate list until a matching pair is found. This matching pair is then returned as the key and certificate, optionally the certificate list found is returned as a STACK_OF certificates. At the moment a CRL, Secret or SafeContents bag will be skipped over and not parsed. It can be seen if these or other “Unknown” bags are skipped over by viewing the debug print out. Additional attributes such as friendly name are skipped over when parsing a PKCS12 file.

A.3.2 Functions Documentation

```
int wolfSSL_BN_mod_exp(
    WOLFSSL_BIGNUM * r,
    const WOLFSSL_BIGNUM * a,
    const WOLFSSL_BIGNUM * p,
    const WOLFSSL_BIGNUM * m,
    WOLFSSL_BN_CTX * ctx
)
```

This function performs the following math “r = (a^p) % m”.

Parameters:

- **r** structure to hold result.
- **a** value to be raised by a power.
- **p** power to raise a by.
- **m** modulus to use.
- **ctx** currently not used with wolfSSL can be NULL.

See:

- wolfSSL_BN_new
- wolfSSL_BN_free

Return:

- SSL_SUCCESS On successfully performing math operation.
- SSL_FAILURE If an error case was encountered.

Example

```
WOLFSSL_BIGNUM r,a,p,m;
int ret;
// set big number values
ret  = wolfSSL_BN_mod_exp(r, a, p, m, NULL);
// check ret value
```

```
const WOLFSSL_EVP_CIPHER * wolfSSL_EVP_des_ede3_ecb(
    void
)
```

Getter functions for the respective WOLFSSL_EVP_CIPHER pointers. wolfSSL_EVP_init() must be called once in the program first to populate these cipher strings. WOLFSSL_DES_ECB macro must be defined for wolfSSL_EVP_des_ede3_ecb().

Parameters:

- **none** No parameters.

See: wolfSSL_EVP_CIPHER_CTX_init

Return: pointer Returns a WOLFSSL_EVP_CIPHER pointer for DES EDE3 operations.

Example

```
printf("block size des ede3 cbc = %d\n",
wolfSSL_EVP_CIPHER_block_size(wolfSSL_EVP_des_ede3_cbc()));
printf("block size des ede3 ecb = %d\n",
wolfSSL_EVP_CIPHER_block_size(wolfSSL_EVP_des_ede3_ecb()));
```

```
const WOLFSSL_EVP_CIPHER * wolfSSL_EVP_des_cbc(
    void
)
```

Getter functions for the respective WOLFSSL_EVP_CIPHER pointers. wolfSSL_EVP_init() must be called once in the program first to populate these cipher strings. WOLFSSL_DES_ECB macro must be defined for wolfSSL_EVP_des_ecb().

Parameters:

- **none** No parameters.

See: wolfSSL_EVP_CIPHER_CTX_init

Return: pointer Returns a WOLFSSL_EVP_CIPHER pointer for DES operations.

Example

```
WOLFSSL_EVP_CIPHER* cipher;
cipher = wolfSSL_EVP_des_cbc();
…
```

```
int wolfSSL_EVP_DigestInit_ex(
    WOLFSSL_EVP_MD_CTX * ctx,
    const WOLFSSL_EVP_MD * type,
    WOLFSSL_ENGINE * impl
)
```

Function for initializing WOLFSSL_EVP_MD_CTX. This function is a wrapper for wolfSSL_EVP_DigestInit() because wolfSSL does not use WOLFSSL_ENGINE.

Parameters:

- **ctx** structure to initialize.
- **type** type of hash to do, for example SHA.
- **impl** engine to use. N/A for wolfSSL, can be NULL.

See:

- wolfSSL_EVP_MD_CTX_new
- wolfCrypt_Init
- wolfSSL_EVP_MD_CTX_free

Return:

- SSL_SUCCESS If successfully set.
- SSL_FAILURE If not successful.

Example

```
WOLFSSL_EVP_MD_CTX* md = NULL;
wolfCrypt_Init();
md = wolfSSL_EVP_MD_CTX_new();
if (md == NULL) {
    printf("error setting md\n");
    return -1;
}
printf("cipher md init ret = %d\n", wolfSSL_EVP_DigestInit_ex(md,
wolfSSL_EVP_sha1(), e));
//free resources
```

```
int wolfSSL_EVP_CipherInit_ex(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    const WOLFSSL_EVP_CIPHER * type,
    WOLFSSL_ENGINE * impl,
    const unsigned char * key,
    const unsigned char * iv,
    int enc
)
```

Function for initializing WOLFSSL_EVP_CIPHER_CTX. This function is a wrapper for wolfSSL_CipherInit() because wolfSSL does not use WOLFSSL_ENGINE.

Parameters:

- **ctx** structure to initialize.
- **type** type of encryption/decryption to do, for example AES.
- **impl** engine to use. N/A for wolfSSL, can be NULL.
- **key** key to set .
- **iv** iv if needed by algorithm.
- **enc** encryption (1) or decryption (0) flag.

See:

- wolfSSL_EVP_CIPHER_CTX_new
- wolfCrypt_Init
- wolfSSL_EVP_CIPHER_CTX_free

Return:

- SSL_SUCCESS If successfully set.
- SSL_FAILURE If not successful.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx = NULL;
WOLFSSL_ENGINE* e = NULL;
unsigned char key[16];
unsigned char iv[12];
wolfCrypt_Init();
ctx = wolfSSL_EVP_CIPHER_CTX_new();
if (ctx == NULL) {
    printf("issue creating ctx\n");
    return -1;
}

printf("cipher init ex error ret = %d\n", wolfSSL_EVP_CipherInit_ex(NULL,
EVP_aes_128_    cbc(), e, key, iv, 1));
printf("cipher init ex success ret = %d\n", wolfSSL_EVP_CipherInit_ex(ctx,
EVP_aes_128_c    bc(), e, key, iv, 1));
// free resources
```

```
int wolfSSL_EVP_EncryptInit_ex(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    const WOLFSSL_EVP_CIPHER * type,
    WOLFSSL_ENGINE * impl,
    const unsigned char * key,
    const unsigned char * iv
)
```

Function for initializing WOLFSSL_EVP_CIPHER_CTX. This function is a wrapper for wolfSSL_EVP_CipherInit() because wolfSSL does not use WOLFSSL_ENGINE. Sets encrypt flag to be encrypt.

Parameters:

- **ctx** structure to initialize.
- **type** type of encryption to do, for example AES.
- **impl** engine to use. N/A for wolfSSL, can be NULL.
- **key** key to use.
- **iv** iv to use.

See:

- wolfSSL_EVP_CIPHER_CTX_new
- wolfCrypt_Init
- wolfSSL_EVP_CIPHER_CTX_free

Return:

- SSL_SUCCESS If successfully set.
- SSL_FAILURE If not successful.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx = NULL;
wolfCrypt_Init();
ctx = wolfSSL_EVP_CIPHER_CTX_new();
if (ctx == NULL) {
    printf("error setting ctx\n");
    return -1;
}
printf("cipher ctx init ret = %d\n", wolfSSL_EVP_EncryptInit_ex(ctx,
wolfSSL_EVP_aes_128_cbc(), e, key, iv));
//free resources
```

```
int wolfSSL_EVP_DecryptInit_ex(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    const WOLFSSL_EVP_CIPHER * type,
    WOLFSSL_ENGINE * impl,
    const unsigned char * key,
    const unsigned char * iv
)
```

Function for initializing WOLFSSL_EVP_CIPHER_CTX. This function is a wrapper for wolfSSL_EVP_CipherInit() because wolfSSL does not use WOLFSSL_ENGINE. Sets encrypt flag to be decrypt.

Parameters:

- **ctx** structure to initialize.
- **type** type of encryption/decryption to do, for example AES.
- **impl** engine to use. N/A for wolfSSL, can be NULL.
- **key** key to set .
- **iv** iv if needed by algorithm.
- **enc** encryption (1) or decryption (0) flag.

See:

- wolfSSL_EVP_CIPHER_CTX_new
- wolfCrypt_Init
- wolfSSL_EVP_CIPHER_CTX_free

Return:

- SSL_SUCCESS If successfully set.
- SSL_FAILURE If not successful.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx = NULL;
WOLFSSL_ENGINE* e = NULL;
unsigned char key[16];
unsigned char iv[12];

wolfCrypt_Init();

ctx = wolfSSL_EVP_CIPHER_CTX_new();
if (ctx == NULL) {
    printf("issue creating ctx\n");
    return -1;
}

printf("cipher init ex error ret = %d\n", wolfSSL_EVP_DecryptInit_ex(NULL,
EVP_aes_128_    cbc(), e, key, iv, 1));
printf("cipher init ex success ret = %d\n", wolfSSL_EVP_DecryptInit_ex(ctx,
EVP_aes_128_c    bc(), e, key, iv, 1));
// free resources
```

```
int wolfSSL_EVP_CipherUpdate(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    unsigned char * out,
    int * outl,
    const unsigned char * in,
    int inl
)
```

Function for encrypting/decrypting data. In buffer is added to be encrypted or decrypted and out buffer holds the results. outl will be the length of encrypted/decrypted information.

Parameters:

- **ctx** structure to get cipher type from.
- **out** buffer to hold output, at least inl plus the cipher block size bytes. It must not overlap in.
- **outl** adjusted to be size of output.
- **in** buffer to perform operation on.
- **inl** length of input buffer.

See:

- wolfSSL_EVP_CIPHER_CTX_new
- wolfCrypt_Init
- wolfSSL_EVP_CIPHER_CTX_free

Return:

- SSL_SUCCESS If successful.
- SSL_FAILURE If not successful.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx = NULL;
unsigned char out[100 + EVP_MAX_BLOCK_LENGTH];
int outl;
unsigned char in[100];
int inl = 100;

ctx = wolfSSL_EVP_CIPHER_CTX_new();
// set up ctx
ret = wolfSSL_EVP_CipherUpdate(ctx, out, outl, in, inl);
// check ret value
// buffer out holds outl bytes of data
// free resources
```

```
int wolfSSL_EVP_CipherFinal(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    unsigned char * out,
    int * outl
)
```

This function performs the final cipher operations adding in padding. If WOLFSSL_EVP_CIPH_NO_PADDING flag is set in WOLFSSL_EVP_CIPHER_CTX structure then 1 is returned and no encryption/decryption is done. If padding flag is seti padding is added and encrypted when ctx is set to encrypt, padding values are checked when set to decrypt.

Parameters:

- **ctx** structure to decrypt/encrypt with.
- **out** buffer for final decrypt/encrypt.
- **out1** size of out buffer when data has been added by function.

See: wolfSSL_EVP_CIPHER_CTX_new

Return:

- 1 Returned on success.
- 0 If encountering a failure.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
int out1;
unsigned char out[64];
// create ctx
wolfSSL_EVP_CipherFinal(ctx, out, &out1);
```

```
int wolfSSL_EVP_CIPHER_CTX_set_key_length(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    int keylen
)
```

Setter function for WOLFSSL_EVP_CIPHER_CTX structure key length.

Parameters:

- **ctx** structure to set key length.
- **keylen** key length.

See: wolfSSL_EVP_CIPHER_flags

Return:

- SSL_SUCCESS If successfully set.
- SSL_FAILURE If failed to set key length.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
int keylen;
// create ctx
wolfSSL_EVP_CIPHER_CTX_set_key_length(ctx, keylen);
```

```
int wolfSSL_EVP_CIPHER_CTX_block_size(
    const WOLFSSL_EVP_CIPHER_CTX * ctx
)
```

This is a getter function for the ctx block size.

Parameters:

- **ctx** the cipher ctx to get block size of.

See: wolfSSL_EVP_CIPHER_block_size

Return: size Returns ctx->block_size.

Example

```
const WOLFSSL_CVP_CIPHER_CTX* ctx;
//set up ctx
printf("block size = %d\n", wolfSSL_EVP_CIPHER_CTX_block_size(ctx));
```

```
int wolfSSL_EVP_CIPHER_block_size(
    const WOLFSSL_EVP_CIPHER * cipher
)
```

This is a getter function for the block size of cipher.

Parameters:

- **cipher** cipher to get block size of.

See: wolfSSL_EVP_aes_256_ctr

Return: size returns the block size.

Example

```
printf("block size = %d\n",
wolfSSL_EVP_CIPHER_block_size(wolfSSL_EVP_aes_256_ecb()));
```

```
void wolfSSL_EVP_CIPHER_CTX_set_flags(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    int flags
)
```

Setter function for WOLFSSL_EVP_CIPHER_CTX structure.

Parameters:

- **ctx** structure to set flag.
- **flag** flag to set in structure.

See:

- wolfSSL_EVP_CIPHER_flags
- wolfSSL_EVP_CIPHER_CTX_flags

Return: none No returns.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
int flag;
// create ctx
wolfSSL_EVP_CIPHER_CTX_set_flags(ctx, flag);
```

```
void wolfSSL_EVP_CIPHER_CTX_clear_flags(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    int flags
)
```

Clearing function for WOLFSSL_EVP_CIPHER_CTX structure.

Parameters:

- **ctx** structure to clear flag.
- **flag** flag value to clear in structure.

See:

- wolfSSL_EVP_CIPHER_flags
- wolfSSL_EVP_CIPHER_CTX_flags

Return: none No returns.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
int flag;
// create ctx
wolfSSL_EVP_CIPHER_CTX_clear_flags(ctx, flag);
```

```
int wolfSSL_EVP_CIPHER_CTX_set_padding(
    WOLFSSL_EVP_CIPHER_CTX * c,
    int pad
)
```

Setter function for WOLFSSL_EVP_CIPHER_CTX structure to use padding.

Parameters:

- **ctx** structure to set padding flag.
- **padding** 0 for not setting padding, 1 for setting padding.

See: wolfSSL_EVP_CIPHER_CTX_new

Return:

- SSL_SUCCESS If successfully set.
- BAD_FUNC_ARG If null argument passed in.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
// create ctx
wolfSSL_EVP_CIPHER_CTX_set_padding(ctx, 1);
```

```
unsigned long wolfSSL_EVP_CIPHER_CTX_flags(
    const WOLFSSL_EVP_CIPHER_CTX * ctx
)
```

Getter function for WOLFSSL_EVP_CIPHER_CTX structure. Deprecated v1.1.0.

Parameters:

- **ctx** structure to get flag.

See:

- wolfSSL_EVP_CIPHER_CTX_new
- wolfSSL_EVP_CIPHER_flags

Return: unsigned long of flags/mode.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
unsigned long flags;
ctx = wolfSSL_EVP_CIPHER_CTX_new()
flags = wolfSSL_EVP_CIPHER_CTX_flags(ctx);
```

```
int wolfSSL_PEM_write_bio_PrivateKey(
    WOLFSSL_BIO * bio,
    WOLFSSL_EVP_PKEY * key,
    const WOLFSSL_EVP_CIPHER * cipher,
    unsigned char * passwd,
    int len,
    wc_pem_password_cb * cb,
    void * arg
)
```

This function writes a key into a WOLFSSL_BIO structure in PEM format.

Parameters:

- **bio** WOLFSSL_BIO structure to get PEM buffer from.
- **key** key to convert to PEM format.
- **cipher** EVP cipher structure.
- **passwd** password.
- **len** length of password.
- **cb** password callback.
- **arg** optional argument.

See: wolfSSL_PEM_read_bio_X509_AUX

Return:

- SSL_SUCCESS upon success.
- SSL_FAILURE upon failure.

Example

```
WOLFSSL_BIO* bio;
WOLFSSL_EVP_PKEY* key;
int ret;
// create bio and setup key
```

```
ret = wolfSSL_PEM_write_bio_PrivateKey(bio, key, NULL, NULL, 0, NULL, NULL);
//check ret value
```

```
int wolfSSL_CTX_use_RSAPrivateKey_file(
    WOLFSSL_CTX * ctx,
    const char * file,
    int format
)
```

This function loads the private RSA key used in the SSL connection into the SSL context (WOLFSSL_CTX). This function is only available when wolfSSL has been compiled with the OpenSSL compatibility layer enabled (–enable-opensslExtra, #define OPENSSL_EXTRA), and is identical to the more-typically used wolfSSL_CTX_use_PrivateKey_file() function. The file argument contains a pointer to the RSA private key file, in the format specified by format.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new()
- **file** a pointer to the name of the file containing the RSA private key to be loaded into the wolfSSL SSL context, with format as specified by format.
- **format** the encoding type of the RSA private key specified by file. Possible values include SSL_FILETYPE_PEM and SSL_FILETYPE_ASN1.

See:

- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_PrivateKey_file
- wolfSSL_use_RSAPrivateKey_file
- wolfSSL_use_PrivateKey_buffer
- wolfSSL_use_PrivateKey_file

Return:

- SSL_SUCCESS upon success.
- SSL_FAILURE If the function call fails, possible causes might include: The input key file is in the wrong format, or the wrong format has been given using the “format” argument, file doesn’t exist, can’t be read, or is corrupted, an out of memory condition occurs.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_use_RSAPrivateKey_file(ctx, “./server-key.pem”,
                                         SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading private key file
}
...
```

```
int wolfSSL_use_certificate_file(
    WOLFSSL * ssl,
    const char * file,
    int format
)
```

This function loads a certificate file into the SSL session (WOLFSSL structure). The certificate file is provided by the file argument. The format argument specifies the format type of the file - either SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created with wolfSSL_new().
- **file** a pointer to the name of the file containing the certificate to be loaded into the wolfSSL SSL session, with format as specified by format.
- **format** the encoding type of the certificate specified by file. Possible values include SSL_FILETYPE_PEM and SSL_FILETYPE_ASN1.

See:

- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_CTX_use_certificate_file
- wolfSSL_use_certificate_buffer

Return:

- SSL_SUCCESS upon success
- SSL_FAILURE If the function call fails, possible causes might include: The file is in the wrong format, or the wrong format has been given using the “format” argument, file doesn’t exist, can’t be read, or is corrupted, an out of memory condition occurs, Base16 decoding fails on the file

Example

```
int ret = 0;
WOLFSSL* ssl;
...
ret = wolfSSL_use_certificate_file(ssl, “./client-cert.pem”,
                                   SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading cert file
}
...
```

```
int wolfSSL_use_PrivateKey_file(
    WOLFSSL * ssl,
    const char * file,
    int format
)
```

This function loads a private key file into the SSL session (WOLFSSL structure). The key file is provided by the file argument. The format argument specifies the format type of the file - SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created with wolfSSL_new().
- **file** a pointer to the name of the file containing the key file to be loaded into the wolfSSL SSL session, with format as specified by format.
- **format** the encoding type of the key specified by file. Possible values include SSL_FILETYPE_PEM and SSL_FILETYPE_ASN1.

See:

- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_PrivateKey_file
- wolfSSL_use_PrivateKey_buffer
- wc_CryptoCb_RegisterDevice
- wolfSSL_SetDevId

Return:

- SSL_SUCCESS upon success.
- SSL_FAILURE If the function call fails, possible causes might include: The file is in the wrong format, or the wrong format has been given using the “format” argument, The file doesn’t exist, can’t be read, or is corrupted, An out of memory condition occurs, Base16 decoding fails on the file, The key file is encrypted but no password is provided

If using an external key store and do not have the private key you can instead provide the public key and register the crypro callback to handle the signing. For this you can build with either build with crypto callbacks or PK callbacks. To enable crypto callbacks use –enable-cryptocb or WOLF_CRYPTO_CB and register a crypto callback using wc_CryptoCb_RegisterDevice and set the associated devId using wolfSSL_SetDevId.

Example

```
int ret = 0;
WOLFSSL* ssl;
...
ret = wolfSSL_use_PrivateKey_file(ssl, “./server-key.pem”,
                                  SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading key file
}
...
```

```
int wolfSSL_use_certificate_chain_file(
    WOLFSSL * ssl,
    const char * file
)
```

This function loads a chain of certificates into the SSL session (WOLFSSL structure). The file containing the certificate chain is provided by the file argument, and must contain PEM-formatted certificates. This function will process up to MAX_CHAIN_DEPTH (default = 9, defined in internal.h) certificates, plus the subject certificate.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new()
- **file** a pointer to the name of the file containing the chain of certificates to be loaded into the wolfSSL SSL session. Certificates must be in PEM format.

See:

- wolfSSL_CTX_use_certificate_chain_file
- wolfSSL_CTX_use_certificate_chain_buffer
- wolfSSL_use_certificate_chain_buffer

Return:

- SSL_SUCCESS upon success.
- SSL_FAILURE If the function call fails, possible causes might include: The file is in the wrong format, or the wrong format has been given using the “format” argument, file doesn’t exist, can’t be read, or is corrupted, an out of memory condition occurs

Example

```
int ret = 0;
WOLFSSL* ctx;
...
ret = wolfSSL_use_certificate_chain_file(ssl, “./cert-chain.pem”);
if (ret != SSL_SUCCESS) {
    // error loading cert file
}
...
```

```
int wolfSSL_use_RSAPrivateKey_file(
    WOLFSSL * ssl,
    const char * file,
    int format
)
```

This function loads the private RSA key used in the SSL connection into the SSL session (WOLFSSL structure). This function is only available when wolfSSL has been compiled with the OpenSSL compatibility layer enabled (–enable-opensslExtra, #define OPENSSL_EXTRA), and is identical to the more-typically used wolfSSL_use_PrivateKey_file() function. The file argument contains a pointer to the RSA private key file, in the format specified by format.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new()

- **file** a pointer to the name of the file containing the RSA private key to be loaded into the wolfSSL SSL session, with format as specified by format.
- **format** the encoding type of the RSA private key specified by file. Possible values include SSL_FILETYPE_PEM and SSL_FILETYPE_ASN1.

See:

- wolfSSL_CTX_use_RSAPrivateKey_file
- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_PrivateKey_file
- wolfSSL_use_PrivateKey_buffer
- wolfSSL_use_PrivateKey_file

Return:

- SSL_SUCCESS upon success
- SSL_FAILURE If the function call fails, possible causes might include: The input key file is in the wrong format, or the wrong format has been given using the “format” argument, file doesn’t exist, can’t be read, or is corrupted, an out of memory condition occurs

Example

```
int ret = 0;
WOLFSSL* ssl;
...
ret = wolfSSL_use_RSAPrivateKey_file(ssl, “./server-key.pem”,
                                     SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading private key file
}
...
```

```
long wolfSSL_set_tlsext_status_type(
    WOLFSSL * s,
    int type
)
```

This function is called when the client application request that a server send back an OCSP status response (also known as OCSP stapling).Currently, the only supported type is TLSEXT_STATUSTYPE_ocsp.

Parameters:

- **s** pointer to WOLFSSL struct which is created by SSL_new() function
- **type** ssl extension type which TLSEXT_STATUSTYPE_ocsp is only supported.

See:

- wolfSSL_new
- wolfSSL_CTX_new
- wolfSSL_free

- wolfSSL_CTX_free

Return:

- 1 upon success.
- 0 upon error.
- BAD_FUNC_ARG will be returned when s is NULL.
- SSL_FAILURE will be returned when the type is not OCSP.

Example

```
WOLFSSL *ssl;
WOLFSSL_CTX *ctx;
int ret;
ctx = wolfSSL_CTX_new(wolfSSLv23_server_method());
ssl = wolfSSL_new(ctx);
ret = WolfSSL_set_tlsext_status_type(ssl,TLSEXT_STATUSTYPE_ocsp);
wolfSSL_free(ssl);
wolfSSL_CTX_free(ctx);
```

```
WOLFSSL_X509_CHAIN * wolfSSL_get_peer_chain(
    WOLFSSL * ssl
)
```

Retrieves the peer's certificate chain.

Parameters:

- **ssl** pointer to a valid WOLFSSL structure.

See:

- wolfSSL_get_chain_count
- wolfSSL_get_chain_length
- wolfSSL_get_chain_cert
- wolfSSL_get_chain_cert_pem

Return:

- chain If successful the call will return the peer's certificate chain.
- 0 will be returned if an invalid WOLFSSL pointer is passed to the function.

Example

none

```
int wolfSSL_get_chain_count(
    WOLFSSL_X509_CHAIN * chain
)
```

Retrieve's the peers certificate chain count.

Parameters:

- **chain** pointer to a valid WOLFSSL_X509_CHAIN structure.

See:

- wolfSSL_get_peer_chain
- wolfSSL_get_chain_length
- wolfSSL_get_chain_cert
- wolfSSL_get_chain_cert_pem

Return:

- Success If successful the call will return the peer's certificate chain count.
- 0 will be returned if an invalid chain pointer is passed to the function.

Example

none

```
int wolfSSL_get_chain_length(
    WOLFSSL_X509_CHAIN * chain,
    int idx
)
```

Retrieves the peer's ASN1.DER certificate length in bytes at index (idx).

Parameters:

- **chain** pointer to a valid WOLFSSL_X509_CHAIN structure.
- **idx** index to start of chain.

See:

- wolfSSL_get_peer_chain
- wolfSSL_get_chain_count
- wolfSSL_get_chain_cert
- wolfSSL_get_chain_cert_pem

Return:

- Success If successful the call will return the peer's certificate length in bytes by index.

- 0 will be returned if an invalid chain pointer is passed to the function, or if the slot at idx holds no certificate because the one presented there was too large to store.

Example

none

```
unsigned char * wolfSSL_get_chain_cert(
    WOLFSSL_X509_CHAIN * chain,
    int idx
)
```

Retrieves the peer's ASN1.DER certificate at index (idx).

Parameters:

- **chain** pointer to a valid WOLFSSL_X509_CHAIN structure.
- **idx** index to start of chain.

See:

- wolfSSL_get_peer_chain
- wolfSSL_get_chain_count
- wolfSSL_get_chain_length
- wolfSSL_get_chain_cert_pem

Return:

- Success If successful the call will return the peer's certificate by index.
- 0 will be returned if an invalid chain pointer is passed to the function, or if the slot at idx holds no certificate because the one presented there was too large to store. Indices inside wolfSSL_get_chain_count() can be empty, so check wolfSSL_get_chain_length() before using the result.

Example

none

```
int wolfSSL_get_chain_cert_pem(
    WOLFSSL_X509_CHAIN * chain,
    int idx,
    unsigned char * buf,
    int inLen,
    int * outLen
)
```

Retrieves the peer's PEM certificate at index (idx).

Parameters:

- **chain** pointer to a valid WOLFSSL_X509_CHAIN structure.
- **idx** index to start of chain.
- **buf** buffer to hold the PEM certificate. May be NULL to ask for the required length only.
- **inLen** length of buf in bytes.
- **outLen** length of the PEM data in bytes.

See:

- wolfSSL_get_peer_chain
- wolfSSL_get_chain_count
- wolfSSL_get_chain_length
- wolfSSL_get_chain_cert

Return:

- SSL_SUCCESS will be returned on success.
- SSL_FAILURE will be returned when the certificate cannot be converted.
- BAD_FUNC_ARG will be returned when chain is NULL, idx is out of range, outLen is NULL, or inLen is negative. An idx inside wolfSSL_get_chain_count() whose slot holds no certificate, because the one presented there was too large to store, is reported the same way.
- BUFFER_E will be returned when inLen is too small to hold the PEM output, in a build that converts with wc_DerToPem(). A build that has only WOLFSSL_PEM_TO_DER reports that case as BAD_FUNC_ARG, so a caller growing its buffer on a short write should accept either.
- LENGTH_ONLY_E will be returned when buf is NULL, with the required length returned through outLen.

Example

none

```
const unsigned char * wolfSSL_get_sessionID(
    const WOLFSSL_SESSION * s
)
```

Retrieves the session's ID. The session ID is always 32 bytes long.

Parameters:

- **session** pointer to a valid wolfssl session.

See: SSL_get_session

Return: id The session ID.

Example

none

```
int wolfSSL_X509_get_serial_number(
    WOLFSSL_X509 * x509,
    unsigned char * in,
    int * inOutSz
)
```

Retrieves the peer's certificate serial number. The serial number buffer (in) should be at least 32 bytes long and be provided as the *inOutSz argument as input. After calling the function* inOutSz will hold the actual length in bytes written to the in buffer.

Parameters:

- **in** The serial number buffer and should be at least 32 bytes long
- **inOutSz** will hold the actual length in bytes written to the in buffer.

See: SSL_get_peer_certificate

Return:

- SSL_SUCCESS upon success.
- BAD_FUNC_ARG will be returned if a bad function argument was encountered.

Example

none

```
WC_PKCS12 * wolfSSL_d2i_PKCS12_bio(
    WOLFSSL_BIO * bio,
    WC_PKCS12 ** pkcs12
)
```

wolfSSL_d2i_PKCS12_bio (d2i_PKCS12_bio) copies in the PKCS12 information from WOLFSSL_BIO to the structure WC_PKCS12. The information is divided up in the structure as a list of Content Infos along with a structure to hold optional MAC information. After the information has been divided into chunks (but not decrypted) in the structure WC_PKCS12, it can then be parsed and decrypted by calling.

Parameters:

- **bio** WOLFSSL_BIO structure to read PKCS12 buffer from.
- **pkcs12** WC_PKCS12 structure pointer for new PKCS12 structure created. Can be NULL

See:

- wolfSSL_PKCS12_parse
- wc_PKCS12_free

Return:

- WC_PKCS12 pointer to a WC_PKCS12 structure.
- Failure If function failed it will return NULL.

Example

```
WC_PKCS12* pkcs;
WOLFSSL_BIO* bio;
WOLFSSL_X509* cert;
WOLFSSL_EVP_PKEY* pkey;
STACK_OF(X509) certs;
//bio loads in PKCS12 file
wolfSSL_d2i_PKCS12_bio(bio, &pkcs);
wolfSSL_PKCS12_parse(pkcs, “a password”, &pkey, &cert, &certs)
wc_PKCS12_free(pkcs)
//use cert, pkey, and optionally certs stack
```

```
WC_PKCS12 * wolfSSL_i2d_PKCS12_bio(
    WOLFSSL_BIO * bio,
    WC_PKCS12 * pkcs12
)
```

wolfSSL_i2d_PKCS12_bio (i2d_PKCS12_bio) copies in the cert information from the structure WC_PKCS12 to WOLFSSL_BIO.

Parameters:

- **bio** WOLFSSL_BIO structure to write PKCS12 buffer to.
- **pkcs12** WC_PKCS12 structure for PKCS12 structure as input.

See:

- wolfSSL_PKCS12_parse
- wc_PKCS12_free

Return:

- 1 for success.
- Failure 0.

Example

```
WC_PKCS12 pkcs12;
FILE *f;
byte buffer[5300];
char file[] = "./test.p12";
int bytes;
WOLFSSL_BIO* bio;
```

```
pkcs12 = wc_PKCS12_new();
f = fopen(file, "rb");
bytes = (int)fread(buffer, 1, sizeof(buffer), f);
fclose(f);
//convert the DER file into an internal structure
wc_d2i_PKCS12(buffer, bytes, pkcs12);
bio = wolfSSL_BIO_new(wolfSSL_BIO_s_mem());
//convert PKCS12 structure into bio
wolfSSL_i2d_PKCS12_bio(bio, pkcs12);
wc_PKCS12_free(pkcs)
//use bio
```

```
int wolfSSL_PKCS12_parse(
    WC_PKCS12 * pkcs12,
    const char * psw,
    WOLFSSL_EVP_PKEY ** pkey,
    WOLFSSL_X509 ** cert,
    WOLF_STACK_OF(WOLFSSL_X509) ** ca
)
```

PKCS12 can be enabled with adding –enable-opensslextra to the configure command. It can use triple DES and RC4 for decryption so would recommend also enabling these features when enabling opensslextra (–enable-des3 –enable-arc4). wolfSSL does not currently support RC2 so decryption with RC2 is currently not available. This may be noticeable with default encryption schemes used by OpenSSL command line to create .p12 files. wolfSSL_PKCS12_parse (PKCS12_parse). The first thing this function does is check the MAC is correct if present. If the MAC fails then the function returns and does not try to decrypt any of the stored Content Infos. This function then parses through each Content Info looking for a bag type, if the bag type is known it is decrypted as needed and either stored in the list of certificates being built or as a key found. After parsing through all bags the key found is then compared with the certificate list until a matching pair is found. This matching pair is then returned as the key and certificate, optionally the certificate list found is returned as a STACK_OF certificates. At the moment a CRL, Secret or SafeContents bag will be skipped over and not parsed. It can be seen if these or other “Unknown” bags are skipped over by viewing the debug print out. Additional attributes such as friendly name are skipped over when parsing a PKCS12 file.

Parameters:

- **pkcs12** WC_PKCS12 structure to parse.
- **paswd** password for decrypting PKCS12.
- **pkey** structure to hold private key decoded from PKCS12.
- **cert** structure to hold certificate decoded from PKCS12.
- **stack** optional stack of extra certificates.

See:

- wolfSSL_d2i_PKCS12_bio
- wc_PKCS12_free

Return:

- SSL_SUCCESS On successfully parsing PKCS12.

- SSL_FAILURE If an error case was encountered.

Example

```
WC_PKCS12* pkcs;
WOLFSSL_BIO* bio;
WOLFSSL_X509* cert;
WOLFSSL_EVP_PKEY* pkey;
STACK_OF(X509) certs;
//bio loads in PKCS12 file
wolfSSL_d2i_PKCS12_bio(bio, &pkcs);
wolfSSL_PKCS12_parse(pkcs, “a password”, &pkey, &cert, &certs)
wc_PKCS12_free(pkcs)
//use cert, pkey, and optionally certs stack
```

A.4 wolfSSL Certificates and Keys

A.3.2.33 function wolfSSL_PKCS12_parse

A.4.1 Functions

	Name
int	**wc_KeyPemToDer**(const unsigned char * pem, int pemSz, unsigned char * buff, int buffSz, const char * pass)Converts a key in PEM format to DER format.
int	**wc_CertPemToDer**(const unsigned char * pem, int pemSz, unsigned char * buff, int buffSz, int type)This function converts a PEM formatted certificate to DER format. Calls OpenSSL function PemToDer.
int	**wc_GetPubKeyDerFromCert**(struct DecodedCert * cert, byte * derKey, word32 * derKeySz)This function gets the public key in DER format from a populated DecodedCert struct. Users must call wc_InitDecodedCert() and wc_ParseCert() before calling this API. wc_InitDecodedCert() accepts a DER/ASN.1 encoded certificate. To convert a PEM cert to DER, first use wc_CertPemToDer() before calling wc_InitDecodedCert().
int	**wolfSSL_CTX_use_certificate_file**(WOLFSSL_CTX * ctx, const char * file, int format)This function loads a certificate file into the SSL context (WOLFSSL_CTX). The file is provided by the file argument. The format argument specifies the format type of the file, either SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. Please see the examples for proper usage.

	Name
int	**wolfSSL_CTX_use_PrivateKey_file**(WOLFSSL_CTX * ctx, const char * file, int format)This function loads a private key file into the SSL context (WOLFSSL_CTX). The file is provided by the file argument. The format argument specifies the format type of the file - SSL_FILETYPE_ASN1or SSL_FILETYPE_PEM. Please see the examples for proper usage.
int	**wolfSSL_CTX_load_verify_locations**(WOLFSSL_CTX * ctx, const char * file, const char * path)This function loads PEM_formatted CA certificate files into the SSL context (WOLFSSL_CTX). These certificates will be treated as trusted root certificates and used to verify certs received from peers during the SSL handshake. The root certificate file, provided by the file argument, may be a single certificate or a file containing multiple certificates. If multiple CA certs are included in the same file, wolfSSL will load them in the same order they are presented in the file. The path argument is a pointer to the name of a directory that contains certificates of trusted root CAs. If the value of file is not NULL, path may be specified as NULL if not needed. If path is specified and NO_WOLFSSL_DIR was not defined when building the library, wolfSSL will load all CA certificates located in the given directory. This function will attempt to load all files in the directory. This function expects PEM formatted CERT_TYPE file with header “—–BEGIN CERTIFICATE—–”.

	Name
int	**wolfSSL_CTX_load_verify_locations_ex**(WOLFSSL_CTX * ctx, const char * file, const char * path, word32 flags)This function loads PEM_formatted CA certificate files into the SSL context (WOLFSSL_CTX). These certificates will be treated as trusted root certificates and used to verify certs received from peers during the SSL handshake. The root certificate file, provided by the file argument, may be a single certificate or a file containing multiple certificates. If multiple CA certs are included in the same file, wolfSSL will load them in the same order they are presented in the file. The path argument is a pointer to the name of a directory that contains certificates of trusted root CAs. If the value of file is not NULL, path may be specified as NULL if not needed. If path is specified and NO_WOLFSSL_DIR was not defined when building the library, wolfSSL will load all CA certificates located in the given directory. This function will attempt to load all files in the directory based on flags specified. This function expects PEM formatted CERT_TYPE files with header "—–BEGIN CERTIFICATE—–".
const char **	**wolfSSL_get_system_CA_dirs**(word32 * num)This function returns a pointer to an array of strings representing directories wolfSSL will search for system CA certs when wolfSSL_CTX_load_system_CA_certs is called. On systems that don't store certificates in an accessible system directory (such as Apple platforms), this function will always return NULL.
int	**wolfSSL_CTX_load_system_CA_certs**(WOLFSSL_CTX * ctx)On most platforms (including Linux and Windows), this function attempts to load CA certificates into a WOLFSSL_CTX from an OS-dependent CA certificate store. Loaded certificates will be trusted.
int	**wolfSSL_CTX_use_certificate_chain_file**(WOLFSSL_CTX * ctx, const char * file)This function loads a chain of certificates into the SSL context (WOLFSSL_CTX). The file containing the certificate chain is provided by the file argument, and must contain PEM_formatted certificates. This function will process up to MAX_CHAIN_DEPTH (default = 9, defined in internal.h) certificates, plus the subject cert.

	Name
int	**wolfSSL_CTX_der_load_verify_locations**(WOLFSSL_CTX * ctx, const char * file, int format)This function is similar to wolfSSL_CTX_load_verify_locations, but allows the loading of DER_formatted CA files into the SSL context (WOLFSSL_CTX). It may still be used to load PEM_formatted CA files as well. These certificates will be treated as trusted root certificates and used to verify certs received from peers during the SSL handshake. The root certificate file, provided by the file argument, may be a single certificate or a file containing multiple certificates. If multiple CA certs are included in the same file, wolfSSL will load them in the same order they are presented in the file. The format argument specifies the format which the certificates are in either, SSL_FILETYPE_PEM or SSL_FILETYPE_ASN1 (DER). Unlike wolfSSL_CTX_load_verify_locations, this function does not allow the loading of CA certificates from a given directory path. Note that this function is only available when the wolfSSL library was compiled with WOLFSSL_DER_LOAD defined.
void	**wolfSSL_SetCertCbCtx**(WOLFSSL * ssl, void * ctx)This function stores user CTX object information for verify callback.
void	**wolfSSL_CTX_SetCertCbCtx**(WOLFSSL_CTX * ctx, void * userCtx)This function stores user CTX object information for verify callback.

	Name
int	**wolfSSL_CTX_SetChainVerifyCb**(WOLFSSL_CTX * ctx, ChainVerifyCb cb)Replaces wolfSSL's verification of the peer's certificate chain with an application callback, for every SSL/TLS object created from the context. When a callback is set, wolfSSL decodes the certificates from the Certificate message and hands them to the callback as raw DER, the peer's own certificate first. It builds no chain, verifies no signature and checks no date, revocation status, key usage or host name, and the verify callback set with wolfSSL_CTX_set_verify() is not called; the callback is consulted for any certificate the peer sends, even under WOLFSSL_VERIFY_NONE. A server only requests a client certificate under WOLFSSL_VERIFY_PEER and only requires one with WOLFSSL_VERIFY_FAIL_IF_NO_PEER_CERT, so a server that relies on the callback for client authentication must set both. Malformed DER, a certificate the parser refuses regardless of the verify mode such as one with a zero serial number, or one whose decoding stops part way such as one with an unsupported key algorithm, still fails the handshake before the callback is called. An unknown critical extension or a key usage inconsistent with the basic constraints is left for the callback to judge. The callback returns 0 to accept, CHAIN_VERIFY_WANT_E to suspend the handshake until the application re_enters wolfSSL_connect(), wolfSSL_accept(), wolfSSL_read() or wolfSSL_write() (a certificate received after the handshake is resumed by wolfSSL_read() only, and wolfSSL_write() fails with CHAIN_VERIFY_WANT_E until then), or any other value to reject with CHAIN_VERIFY_CB_E and a fatal bad_certificate alert; 0 is the only accepting value, so returning WOLFSSL_SUCCESS rejects. Because wolfSSL checks nothing itself, wolfSSL_get_verify_result() reports WOLFSSL_X509_V_OK once the callback accepts, recording its verdict rather than a verification wolfSSL performed. DTLS, raw public keys and verifying a stapled OCSP response are not supported with the callback: setting it on a context configured for one of them fails, and so does the handshake of a connection using one of them, with CHAIN_VERIFY_UNSUPPORTED_E. The stapling check covers the client, the side that verifies a stapled response; a server may still staple its own status. Requires WOLFSSL_CHAIN_VERIFY_CB (–enable_chain_verify_cb).

	Name
int	**wolfSSL_SetChainVerifyCb**(WOLFSSL * ssl, ChainVerifyCb cb)Sets the chain verification callback for one SSL/TLS object. It takes precedence over the callback set on the context with wolfSSL_CTX_SetChainVerifyCb(), which documents the callback's contract.
void	**wolfSSL_CTX_SetChainVerifyCtx**(WOLFSSL_CTX * ctx, void * userCtx)Sets the user context passed to the chain verification callback for every SSL/TLS object created from the context.
void	**wolfSSL_SetChainVerifyCtx**(WOLFSSL * ssl, void * ctx)Sets the user context passed to the chain verification callback for one SSL/TLS object. It takes precedence over the value set on the context with wolfSSL_CTX_SetChainVerifyCtx().
void *	**wolfSSL_GetChainVerifyCtx**(const WOLFSSL * ssl)Returns the user context the chain verification callback is called with for this SSL/TLS object: the value set with wolfSSL_SetChainVerifyCtx() when there is one, otherwise the context's.
int	**wolfSSL_CTX_save_cert_cache**(WOLFSSL_CTX * ctx, const char * fname)This function writes the cert cache from memory to file.
int	**wolfSSL_CTX_restore_cert_cache**(WOLFSSL_CTX * ctx, const char * fname)This function persistes certificate cache from a file.
int	**wolfSSL_CTX_memsave_cert_cache**(WOLFSSL_CTX * ctx, void * mem, int sz, int * used)This function persists the certificate cache to memory.
int	**wolfSSL_CTX_get_cert_cache_memsize**(WOLFSSL_CTX * ctx)Returns the size the certificate cache save buffer needs to be.
char *	**wolfSSL_X509_NAME_oneline**(WOLFSSL_X509_NAME * name, char * in, int sz)This function copies the name of the x509 into a buffer.
WOLFSSL_X509_NAME *	**wolfSSL_X509_get_issuer_name**(WOLFSSL_X509 * cert)This function returns the name of the certificate issuer.
WOLFSSL_X509_NAME *	**wolfSSL_X509_get_subject_name**(WOLFSSL_X509 * cert)This function returns the subject member of the WOLFSSL_X509 structure.
int	**wolfSSL_X509_get_isCA**(WOLFSSL_X509 * x509)Checks the isCa member of the WOLFSSL_X509 structure and returns the value.
int	**wolfSSL_X509_NAME_get_text_by_NID**(const WOLFSSL_X509_NAME * name, int nid, char * buf, int len)This function gets the text related to the passed in NID value.

	Name
int	**wolfSSL_X509_get_signature_type**(WOLFSSL_X509 * x509)This function returns the value stored in the sigOID member of the WOLFSSL_X509 structure.
int	**wolfSSL_X509_get_signature**(WOLFSSL_X509 * x509, unsigned char * buf, int * bufSz)Gets the X509 signature and stores it in the buffer.
int	**wolfSSL_X509_STORE_add_cert**(WOLFSSL_X509_STORE * store, WOLFSSL_X509 * x509)This function adds a certificate to the WOLFSSL_X509_STRE structure.
WOLFSSL_STACK *	**wolfSSL_X509_STORE_CTX_get_chain**(WOLFSSL_X509_STORE_CT * ctx)This function is a getter function for chain variable in WOLFSSL_X509_STORE_CTX structure. Currently chain is not populated.
int	**wolfSSL_X509_STORE_set_flags**(WOLFSSL_X509_STORE * store, unsigned long flag)This function takes in a flag to change the behavior of the WOLFSSL_X509_STORE structure passed in. An example of a flag used is WOLFSSL_CRL_CHECK. X509_V_FLAG_CRL_CHECK and X509_V_FLAG_CRL_CHECK_ALL enable CRL checking like WOLFSSL_CRL_CHECK and WOLFSSL_CRL_CHECKALL.
const byte *	**wolfSSL_X509_notBefore**(WOLFSSL_X509 * x509)This function the certificate “not before” validity encoded as a byte array.
const byte *	**wolfSSL_X509_notAfter**(WOLFSSL_X509 * x509)This function the certificate “not after” validity encoded as a byte array.
void	**wolfSSL_CTX_set_client_cert_cb**(WOLFSSL_CTX * ctx, client_cert_cb cb)Sets a callback to select the client certificate and private key.
void	**wolfSSL_CTX_set_cert_cb**(WOLFSSL_CTX * ctx, CertSetupCallback cb, void * arg)Sets a generic certificate setup callback.
const char *	**wolfSSL_get_psk_identity_hint**(const WOLFSSL *)This function returns the psk identity hint.
const char *	**wolfSSL_get_psk_identity**(const WOLFSSL *)The function returns a constant pointer to the client_identity member of the Arrays structure.
int	**wolfSSL_CTX_use_psk_identity_hint**(WOLFSSL_CTX * ctx, const char * hint)This function stores the hint argument in the server_hint member of the WOLFSSL_CTX structure.
int	**wolfSSL_use_psk_identity_hint**(WOLFSSL * ssl, const char * hint)This function stores the hint argument in the server_hint member of the Arrays structure within the WOLFSSL structure.
WOLFSSL_X509 *	**wolfSSL_get_peer_certificate**(WOLFSSL * ssl)This function gets the peer’s certificate.

	Name
WOLFSSL_X509 *	**wolfSSL_get_chain_X509**(WOLFSSL_X509_CHAIN * chain, int idx)This function gets the peer's wolfSSL_X509_certificate at index (idx) from the chain of certificates.
char *	**wolfSSL_X509_get_subjectCN**(WOLFSSL_X509 *)Returns the common name of the subject from the certificate.
const unsigned char *	**wolfSSL_X509_get_der**(WOLFSSL_X509 * x509, int * outSz)This function gets the DER encoded certificate in the WOLFSSL_X509 struct.
WOLFSSL_ASN1_TIME *	**wolfSSL_X509_get_notAfter**(WOLFSSL_X509 *)This function checks to see if x509 is NULL and if it's not, it returns the notAfter member of the x509 struct.
int	**wolfSSL_X509_version**(WOLFSSL_X509 * x509)This function retrieves the version of the X509 certificate.
WOLFSSL_X509 *	**wolfSSL_X509_d2i_fp**(WOLFSSL_X509 ** x509, FILE * file)If NO_STDIO_FILESYSTEM is defined this function will allocate heap memory, initialize a WOLFSSL_X509 structure and return a pointer to it.
WOLFSSL_X509 *	**wolfSSL_X509_load_certificate_file**(const char * fname, int format)The function loads the x509 certificate into memory.
unsigned char *	**wolfSSL_X509_get_device_type**(WOLFSSL_X509 * x509, unsigned char * in, int * inOutSz)This function copies the device type from the x509 structure to the buffer.
unsigned char *	**wolfSSL_X509_get_hw_type**(WOLFSSL_X509 * x509, unsigned char * in, int * inOutSz)The function copies the hwType member of the WOLFSSL_X509 structure to the buffer.
unsigned char *	**wolfSSL_X509_get_hw_serial_number**(WOLFSSL_X509 * x509, unsigned char * in, int * inOutSz)This function returns the hwSerialNum member of the x509 object.
int	**wolfSSL_SetTmpDH**(WOLFSSL * ssl, const unsigned char * p, int pSz, const unsigned char * g, int gSz)Server Diffie-Hellman Ephemeral parameters setting. This function sets up the group parameters to be used if the server negotiates a cipher suite that uses DHE.
int	**wolfSSL_SetTmpDH_buffer**(WOLFSSL * ssl, const unsigned char * b, long sz, int format)The function calls the wolfSSL_SetTMpDH_buffer_wrapper, which is a wrapper for Diffie-Hellman parameters.
int	**wolfSSL_SetTmpDH_file**(WOLFSSL * ssl, const char * f, int format)This function calls wolfSSL_SetTmpDH_file_wrapper to set server Diffie-Hellman parameters.

	Name
int	**wolfSSL_CTX_SetTmpDH**(WOLFSSL_CTX * ctx, const unsigned char * p, int pSz, const unsigned char * g, int gSz)Sets the parameters for the server CTX Diffie-Hellman.
int	**wolfSSL_CTX_SetTmpDH_buffer**(WOLFSSL_CTX * ctx, const unsigned char * b, long sz, int format)A wrapper function that calls wolfSSL_SetTmpDH_buffer_wrapper.
int	**wolfSSL_CTX_SetTmpDH_file**(WOLFSSL_CTX * ctx, const char * f, int format)The function calls wolfSSL_SetTmpDH_file_wrapper to set the server Diffie-Hellman parameters.
int	**wolfSSL_CTX_SetMinDhKey_Sz**(WOLFSSL_CTX * ctx, word16 keySz_bits)This function sets the minimum size (in bits) of the Diffie Hellman key size by accessing the minDhKeySz member in the WOLFSSL_CTX structure.
int	**wolfSSL_SetMinDhKey_Sz**(WOLFSSL * ssl, word16 keySz_bits)Sets the minimum size (in bits) for a Diffie-Hellman key in the WOLFSSL structure.
int	**wolfSSL_CTX_SetMaxDhKey_Sz**(WOLFSSL_CTX * ctx, word16 keySz_bits)This function sets the maximum size (in bits) of the Diffie Hellman key size by accessing the maxDhKeySz member in the WOLFSSL_CTX structure.
int	**wolfSSL_SetMaxDhKey_Sz**(WOLFSSL * ssl, word16 keySz_bits)Sets the maximum size (in bits) for a Diffie-Hellman key in the WOLFSSL structure.
int	**wolfSSL_GetDhKey_Sz**(WOLFSSL * ssl)Returns the value of dhKeySz (in bits) that is a member of the options structure. This value represents the Diffie-Hellman key size in bytes.
int	**wolfSSL_CTX_SetMinRsaKey_Sz**(WOLFSSL_CTX * ctx, short keySz)Sets the minimum RSA key size in both the WOLFSSL_CTX structure and the WOLFSSL_CERT_MANAGER structure.
int	**wolfSSL_SetMinRsaKey_Sz**(WOLFSSL * ssl, short keySz)Sets the minimum allowable key size in bits for RSA located in the WOLFSSL structure.
int	**wolfSSL_CTX_SetMinEccKey_Sz**(WOLFSSL_CTX * ctx, short keySz)Sets the minimum size in bits for the ECC key in the WOLF_CTX structure and the WOLFSSL_CERT_MANAGER structure.
int	**wolfSSL_SetMinEccKey_Sz**(WOLFSSL * ssl, short keySz)Sets the value of the minEccKeySz member of the options structure. The options struct is a member of the WOLFSSL structure and is accessed through the ssl parameter.

	Name
int	**wolfSSL_make_eap_keys**(WOLFSSL * ssl, void * key, unsigned int len, const char * label)This function is used by EAP_TLS and EAP-TTLS to derive keying material from the master secret.
int	**wolfSSL_CTX_load_verify_buffer**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format)This function loads a CA certificate buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. More than one CA certificate may be loaded per buffer as long as the format is in PEM. Please see the examples for proper usage.
int	**wolfSSL_CTX_load_verify_buffer_ex**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format, int userChain, word32 flags)This function loads a CA certificate buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. More than one CA certificate may be loaded per buffer as long as the format is in PEM. The _ex version was added in PR 2413 and supports additional arguments for userChain and flags.
int	**wolfSSL_CTX_load_verify_chain_buffer_format**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format)This function loads a CA certificate chain buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. More than one CA certificate may be loaded per buffer as long as the format is in PEM. Please see the examples for proper usage.

	Name
int	**wolfSSL_CTX_use_certificate_buffer**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format)This function loads a certificate buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. Please see the examples for proper usage.
int	**wolfSSL_CTX_use_PrivateKey_buffer**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format)This function loads a private key buffer into the SSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1or SSL_FILETYPE_PEM. Please see the examples for proper usage.
int	**wolfSSL_CTX_use_certificate_chain_buffer**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz)This function loads a certificate chain buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. The buffer must be in PEM format and start with the subject's certificate, ending with the root certificate. Please see the examples for proper usage.
int	**wolfSSL_use_certificate_buffer**(WOLFSSL * ssl, const unsigned char * in, long sz, int format)This function loads a certificate buffer into the WOLFSSL object. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. Please see the examples for proper usage.

	Name
int	**wolfSSL_use_PrivateKey_buffer**(WOLFSSL * ssl, const unsigned char * in, long sz, int format)This function loads a private key buffer into the WOLFSSL object. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. Please see the examples for proper usage.
int	**wolfSSL_use_certificate_chain_buffer**(WOLFSSL * ssl, const unsigned char * in, long sz)This function loads a certificate chain buffer into the WOLFSSL object. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. The buffer must be in PEM format and start with the subject's certificate, ending with the root certificate. Please see the examples for proper usage.
int	**wolfSSL_UnloadCertsKeys**(WOLFSSL * ssl)This function unloads any certificates or keys that SSL owns.
int	**wolfSSL_GetIVSize**(WOLFSSL * ssl)Returns the iv_size member of the specs structure held in the WOLFSSL struct.
void	**wolfSSL_KeepArrays**(WOLFSSL * ssl)Normally, at the end of the SSL handshake, wolfSSL frees temporary arrays. Calling this function before the handshake begins will prevent wolfSSL from freeing temporary arrays. Temporary arrays may be needed for things such as wolfSSL_get_keys() or PSK hints. When the user is done with temporary arrays, either wolfSSL_FreeArrays() may be called to free the resources immediately, or alternatively the resources will be freed when the associated SSL object is freed.
void	**wolfSSL_FreeArrays**(WOLFSSL * ssl)Normally, at the end of the SSL handshake, wolfSSL frees temporary arrays. If wolfSSL_KeepArrays() has been called before the handshake, wolfSSL will not free temporary arrays. This function explicitly frees temporary arrays and should be called when the user is done with temporary arrays and does not want to wait for the SSL object to be freed to free these resources.

	Name
int	**wolfSSL_DeriveTlsKeys**(unsigned char * key_data, word32 keyLen, const unsigned char * ms, word32 msLen, const unsigned char * sr, const unsigned char * cr, int tls1_2, int hash_type)An external facing wrapper to derive TLS Keys.
int	**wolfSSL_X509_get_ext_by_NID**(const WOLFSSL_X509 * x, int nid, int lastpos)This function looks for and returns the extension index matching the passed in NID value.
void *	**wolfSSL_X509_get_ext_d2i**(const WOLFSSL_X509 * x509, int nid, int * c, int * idx)This function looks for and returns the extension matching the passed in NID value.
int	**wolfSSL_X509_digest**(const WOLFSSL_X509 * x509, const WOLFSSL_EVP_MD * digest, unsigned char * buf, unsigned int * len)This function returns the hash of the DER certificate.
int	**wolfSSL_use_PrivateKey**(WOLFSSL * ssl, WOLFSSL_EVP_PKEY * pkey)This is used to set the private key for the WOLFSSL structure.
int	**wolfSSL_use_PrivateKey_ASN1**(int pri, WOLFSSL * ssl, const unsigned char * der, long derSz)This is used to set the private key for the WOLFSSL structure. A DER formatted key buffer is expected.
int	**wolfSSL_use_RSAPrivateKey_ASN1**(WOLFSSL * ssl, unsigned char * der, long derSz)This is used to set the private key for the WOLFSSL structure. A DER formatted RSA key buffer is expected.
WOLFSSL_DH *	**wolfSSL_DSA_dup_DH**(const WOLFSSL_DSA * r)This function duplicates the parameters in dsa to a newly created WOLFSSL_DH structure.
WOLFSSL_X509 *	**wolfSSL_d2i_X509_bio**(WOLFSSL_BIO * bio, WOLFSSL_X509 ** x509)This function get the DER buffer from bio and converts it to a WOLFSSL_X509 structure.
WOLFSSL_X509 *	**wolfSSL_PEM_read_bio_X509_AUX**(WOLFSSL_BIO * bp, WOLFSSL_X509 ** x, wc_pem_password_cb * cb, void * u)This function behaves the same as wolfSSL_PEM_read_bio_X509. AUX signifies containing extra information such as trusted/rejected use cases and friendly name for human readability.
long	**wolfSSL_CTX_set_tmp_dh**(WOLFSSL_CTX * ctx, WOLFSSL_DH * dh)Initializes the WOLFSSL_CTX structure's dh member with the Diffie-Hellman parameters.

	Name
WOLFSSL_DSA *	**wolfSSL_PEM_read_bio_DSAparams**(WOLFSSL_BIO * bp, WOLFSSL_DSA ** x, wc_pem_password_cb * cb, void * u)This function get the DSA parameters from a PEM buffer in bio. **WOLF_STACK_OF**(WOLFSSL_X509) constThis function gets the peer's certificate chain.
char *	**wolfSSL_X509_get_next_altname**(WOLFSSL_X509 *)This function returns the next, if any, altname from the peer certificate.
WOLFSSL_ASN1_TIME *	**wolfSSL_X509_get_notBefore**(WOLFSSL_X509 *)The function checks to see if x509 is NULL and if it's not, it returns the notBefore member of the x509 struct.

A.4.2 Functions Documentation

```
int wc_KeyPemToDer(
    const unsigned char * pem,
    int pemSz,
    unsigned char * buff,
    int buffSz,
    const char * pass
)
```

Converts a key in PEM format to DER format.

Parameters:

- **pem** a pointer to the PEM encoded certificate.
- **pemSz** the size of the PEM buffer (pem)
- **buff** a pointer to the copy of the buffer member of the DerBuffer struct.
- **buffSz** size of the buffer space allocated in the DerBuffer struct.
- **pass** password passed into the function.

See: wc_PemToDer

Return:

- int the function returns the number of bytes written to the buffer on successful execution.
- int negative int returned indicating an error.

Example

```
byte* loadBuf;
long fileSz = 0;
byte* bufSz;
static int LoadKeyFile(byte** keyBuf, word32* keyBufSz,
const char* keyFile,
                int typeKey, const char* password);
…
bufSz = wc_KeyPemToDer(loadBuf, (int)fileSz, saveBuf,
```

```
(int)fileSz, password);

if(saveBufSz > 0){
    // Bytes were written to the buffer.
}
```

```
int wc_CertPemToDer(
    const unsigned char * pem,
    int pemSz,
    unsigned char * buff,
    int buffSz,
    int type
)
```

This function converts a PEM formatted certificate to DER format. Calls OpenSSL function PemToDer.

Parameters:

- **pem** pointer PEM formatted certificate.
- **pemSz** size of the certificate.
- **buff** buffer to be copied to DER format.
- **buffSz** size of the buffer.
- **type** Certificate file type found in asn_public.h enum CertType.

See: wc_PemToDer

Return: buffer returns the bytes written to the buffer.

Example

```
const unsigned char* pem;
int pemSz;
unsigned char buff[BUFSIZE];
int buffSz = sizeof(buff)/sizeof(char);
int type;
...
if(wc_CertPemToDer(pem, pemSz, buff, buffSz, type) <= 0) {
    // There were bytes written to buffer
}
```

```
int wc_GetPubKeyDerFromCert(
    struct DecodedCert * cert,
    byte * derKey,
    word32 * derKeySz
)
```

This function gets the public key in DER format from a populated DecodedCert struct. Users must call wc_InitDecodedCert() and wc_ParseCert() before calling this API. wc_InitDecodedCert() accepts a DER/ASN.1 encoded certificate. To convert a PEM cert to DER, first use wc_CertPemToDer() before calling wc_InitDecodedCert().

Parameters:

- **cert** populated DecodedCert struct holding X.509 certificate
- **derKey** output buffer to place DER encoded public key
- **derKeySz** [IN/OUT] size of derKey buffer on input, size of public key on return. If derKey is passed in as NULL, derKeySz will be set to required buffer size for public key and LENGTH_ONLY_E will be returned from function.

See: wc_GetPubKeyDerFromCert

Return: 0 on success, negative on error. LENGTH_ONLY_E if derKey is NULL and returning length only.

```
int wolfSSL_CTX_use_certificate_file(
    WOLFSSL_CTX * ctx,
    const char * file,
    int format
)
```

This function loads a certificate file into the SSL context (WOLFSSL_CTX). The file is provided by the file argument. The format argument specifies the format type of the file, either SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. Please see the examples for proper usage.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new()
- **file** a pointer to the name of the file containing the certificate to be loaded into the wolfSSL SSL context.
- **format** - format of the certificates pointed to by file. Possible options are SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_use_certificate_file
- wolfSSL_use_certificate_buffer

Return:

- SSL_SUCCESS upon success.
- SSL_FAILURE If the function call fails, possible causes might include the file is in the wrong format, or the wrong format has been given using the “format” argument, file doesn’t exist, can’t be read, or is corrupted, an out of memory condition occurs, Base16 decoding fails on the file.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_use_certificate_file(ctx, “./client-cert.pem”,
                                       SSL_FILETYPE_PEM);
```

```
if (ret != SSL_SUCCESS) {
    // error loading cert file
}
...
```

```
int wolfSSL_CTX_use_PrivateKey_file(
    WOLFSSL_CTX * ctx,
    const char * file,
    int format
)
```

This function loads a private key file into the SSL context (WOLFSSL_CTX). The file is provided by the file argument. The format argument specifies the format type of the file - SSL_FILETYPE_ASN1or SSL_FILETYPE_PEM. Please see the examples for proper usage.

Parameters:

- **ctx** pointer to a WOLFSSL_CTX structure.
- **file** path to the private key file.
- **format** format of the key file (SSL_FILETYPE_PEM or SSL_FILETYPE_ASN1).

See:

- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_use_PrivateKey_file
- wolfSSL_use_PrivateKey_buffer
- wc_CryptoCb_RegisterDevice
- wolfSSL_CTX_SetDevId

Return:

- SSL_SUCCESS upon success.
- SSL_FAILURE The file is in the wrong format, or the wrong format has been given using the "format" argument. The file doesn't exist, can't be read, or is corrupted. An out of memory condition occurs. Base16 decoding fails on the file. The key file is encrypted but no password is provided.

If using an external key store and do not have the private key you can instead provide the public key and register the crypro callback to handle the signing. For this you can build with either build with crypto callbacks or PK callbacks. To enable crypto callbacks use –enable-cryptocb or WOLF_CRYPTO_CB and register a crypto callback using wc_CryptoCb_RegisterDevice and set the associated devId using wolfSSL_CTX_SetDevId.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_use_PrivateKey_file(ctx, "./server-key.pem",
                                      SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading key file
```

```
}
...
```

```
int wolfSSL_CTX_load_verify_locations(
    WOLFSSL_CTX * ctx,
    const char * file,
    const char * path
)
```

This function loads PEM-formatted CA certificate files into the SSL context (WOLFSSL_CTX). These certificates will be treated as trusted root certificates and used to verify certs received from peers during the SSL handshake. The root certificate file, provided by the file argument, may be a single certificate or a file containing multiple certificates. If multiple CA certs are included in the same file, wolfSSL will load them in the same order they are presented in the file. The path argument is a pointer to the name of a directory that contains certificates of trusted root CAs. If the value of file is not NULL, path may be specified as NULL if not needed. If path is specified and NO_WOLFSSL_DIR was not defined when building the library, wolfSSL will load all CA certificates located in the given directory. This function will attempt to load all files in the directory. This function expects PEM formatted CERT_TYPE file with header "—–BEGIN CERTIFICATE—–".

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **file** pointer to name of the file containing PEM-formatted CA certificates.
- **path** pointer to the name of a directory to load PEM-formatted certificates from.

See:

- wolfSSL_CTX_load_verify_locations_ex
- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_use_certificate_file
- wolfSSL_CTX_use_PrivateKey_file
- wolfSSL_CTX_use_certificate_chain_file
- wolfSSL_use_certificate_file
- wolfSSL_use_PrivateKey_file
- wolfSSL_use_certificate_chain_file

Return:

- SSL_SUCCESS up success.
- SSL_FAILURE will be returned if ctx is NULL, or if both file and path are NULL.
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- ASN_BEFORE_DATE_E will be returned if the current date is before the before date.
- ASN_AFTER_DATE_E will be returned if the current date is after the after date.
- BUFFER_E will be returned if a chain buffer is bigger than the receiving buffer.
- BAD_PATH_ERROR will be returned if opendir() fails when trying to open path.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_load_verify_locations(ctx, "./ca-cert.pem", NULL);
if (ret != WOLFSSL_SUCCESS) {
    // error loading CA certs
}
...
```

```
int wolfSSL_CTX_load_verify_locations_ex(
    WOLFSSL_CTX * ctx,
    const char * file,
    const char * path,
    word32 flags
)
```

This function loads PEM-formatted CA certificate files into the SSL context (WOLFSSL_CTX). These certificates will be treated as trusted root certificates and used to verify certs received from peers during the SSL handshake. The root certificate file, provided by the file argument, may be a single certificate or a file containing multiple certificates. If multiple CA certs are included in the same file, wolfSSL will load them in the same order they are presented in the file. The path argument is a pointer to the name of a directory that contains certificates of trusted root CAs. If the value of file is not NULL, path may be specified as NULL if not needed. If path is specified and NO_WOLFSSL_DIR was not defined when building the library, wolfSSL will load all CA certificates located in the given directory. This function will attempt to load all files in the directory based on flags specified. This function expects PEM formatted CERT_TYPE files with header "—–BEGIN CERTIFICATE—–".

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **file** pointer to name of the file containing PEM-formatted CA certificates.
- **path** pointer to the name of a directory to load PEM-formatted certificates from.
- **flags** possible mask values are: WOLFSSL_LOAD_FLAG_IGNORE_ERR, WOLFSSL_LOAD_FLAG_DATE_ERR_OKAY and WOLFSSL_LOAD_FLAG_PEM_CA_ONLY

See:

- wolfSSL_CTX_load_verify_locations
- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_use_certificate_file
- wolfSSL_CTX_use_PrivateKey_file
- wolfSSL_CTX_use_certificate_chain_file
- wolfSSL_use_certificate_file
- wolfSSL_use_PrivateKey_file
- wolfSSL_use_certificate_chain_file

Return:

- SSL_SUCCESS up success.

- SSL_FAILURE will be returned if ctx is NULL, or if both file and path are NULL. This will also be returned if at least one cert is loaded successfully but there is one or more that failed. Check error stack for reason.
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- BUFFER_E will be returned if a chain buffer is bigger than the receiving buffer.
- BAD_PATH_ERROR will be returned if opendir() fails when trying to open path.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_load_verify_locations_ex(ctx, NULL, "./certs/external",
    WOLFSSL_LOAD_FLAG_PEM_CA_ONLY);
if (ret != WOLFSSL_SUCCESS) {
    // error loading CA certs
}
...
```

```
const char ** wolfSSL_get_system_CA_dirs(
    word32 * num
)
```

This function returns a pointer to an array of strings representing directories wolfSSL will search for system CA certs when wolfSSL_CTX_load_system_CA_certs is called. On systems that don't store certificates in an accessible system directory (such as Apple platforms), this function will always return NULL.

Parameters:

- **num** pointer to a word32 that will be populated with the length of the array of strings.

See:

- wolfSSL_CTX_load_system_CA_certs
- wolfSSL_CTX_load_verify_locations
- wolfSSL_CTX_load_verify_locations_ex

Return:

- Valid pointer on success.
- NULL pointer on failure.

Example

```
WOLFSSL_CTX* ctx;
const char** dirs;
word32 numDirs;

dirs = wolfSSL_get_system_CA_dirs(&numDirs);
for (int i = 0; i < numDirs; ++i) {
    printf("Potential system CA dir: %s\n", dirs[i]);
}
...
```

```
int wolfSSL_CTX_load_system_CA_certs(
    WOLFSSL_CTX * ctx
)
```

On most platforms (including Linux and Windows), this function attempts to load CA certificates into a WOLFSSL_CTX from an OS-dependent CA certificate store. Loaded certificates will be trusted.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().

See:

- wolfSSL_get_system_CA_dirs
- wolfSSL_CTX_load_verify_locations
- wolfSSL_CTX_load_verify_locations_ex

Return:

- WOLFSSL_SUCCESS on success.
- WOLFSSL_BAD_PATH if no system CA certs were loaded.
- WOLFSSL_FAILURE for other failure types (e.g. Windows cert store wasn't properly closed).

On Apple platforms (excluding macOS), certificates can't be obtained from the system, and therefore cannot be loaded into the wolfSSL certificate manager. For these platforms, this function enables TLS connections bound to the WOLFSSL_CTX to use the native system trust APIs to verify authenticity of the peer certificate chain if the authenticity of the peer cannot first be authenticated against certificates loaded by the user.

The platforms supported and tested are: Linux (Debian, Ubuntu, Gentoo, Fedora, RHEL), Windows 10/11, Android, macOS, and iOS.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_load_system_CA_certs(ctx,);
if (ret != WOLFSSL_SUCCESS) {
    // error loading system CA certs
```

```
}
...
```

```
int wolfSSL_CTX_use_certificate_chain_file(
    WOLFSSL_CTX * ctx,
    const char * file
)
```

This function loads a chain of certificates into the SSL context (WOLFSSL_CTX). The file containing the certificate chain is provided by the file argument, and must contain PEM-formatted certificates. This function will process up to MAX_CHAIN_DEPTH (default = 9, defined in internal.h) certificates, plus the subject cert.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new()
- **file** a pointer to the name of the file containing the chain of certificates to be loaded into the wolfSSL SSL context. Certificates must be in PEM format.

See:

- wolfSSL_CTX_use_certificate_file
- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_use_certificate_file
- wolfSSL_use_certificate_buffer

Return:

- SSL_SUCCESS upon success
- SSL_FAILURE If the function call fails, possible causes might include the file is in the wrong format, or the wrong format has been given using the “format” argument, file doesn’t exist, can’t be read, or is corrupted, an out of memory condition occurs.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_use_certificate_chain_file(ctx, “./cert-chain.pem”);
if (ret != SSL_SUCCESS) {
    // error loading cert file
}
...
```

```
int wolfSSL_CTX_der_load_verify_locations(
    WOLFSSL_CTX * ctx,
    const char * file,
    int format
)
```

This function is similar to wolfSSL_CTX_load_verify_locations, but allows the loading of DER-formatted CA files into the SSL context (WOLFSSL_CTX). It may still be used to load PEM-formatted CA files as well. These certificates will be treated as trusted root certificates and used to verify certs received from peers during the SSL handshake. The root certificate file, provided by the file argument, may be a single certificate or a file containing multiple certificates. If multiple CA certs are included in the same file, wolfSSL will load them in the same order they are presented in the file. The format argument specifies the format which the certificates are in either, SSL_FILETYPE_PEM or SSL_FILETYPE_ASN1 (DER). Unlike wolfSSL_CTX_load_verify_locations, this function does not allow the loading of CA certificates from a given directory path. Note that this function is only available when the wolfSSL library was compiled with WOLFSSL_DER_LOAD defined.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new()
- **file** a pointer to the name of the file containing the CA certificates to be loaded into the wolfSSL SSL context, with format as specified by format.
- **format** the encoding type of the certificates specified by file. Possible values include SSL_FILETYPE_PEM and SSL_FILETYPE_ASN1.

See:

- wolfSSL_CTX_load_verify_locations
- wolfSSL_CTX_load_verify_buffer

Return:

- SSL_SUCCESS upon success.
- SSL_FAILURE upon failure.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_der_load_verify_locations(ctx, “./ca-cert.der”,
                                            SSL_FILETYPE_ASN1);
if (ret != SSL_SUCCESS) {
    // error loading CA certs
}
...
```

```
void wolfSSL_SetCertCbCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

This function stores user CTX object information for verify callback.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **ctx** a void pointer that is set to WOLFSSL structure's verifyCbCtx member's value.

See:

- wolfSSL_CTX_save_cert_cache
- wolfSSL_CTX_restore_cert_cache
- wolfSSL_CTX_set_verify

Return: none No return.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
(void*)ctx;
...
if(ssl != NULL){
wolfSSL_SetCertCbCtx(ssl, ctx);
} else {
    // Error case, the SSL is not initialized properly.
}
```

```
void wolfSSL_CTX_SetCertCbCtx(
    WOLFSSL_CTX * ctx,
    void * userCtx
)
```

This function stores user CTX object information for verify callback.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure.
- **userCtx** a void pointer that is used to set WOLFSSL_CTX structure's verifyCbCtx member's value.

See:

- wolfSSL_CTX_save_cert_cache
- wolfSSL_CTX_restore_cert_cache
- wolfSSL_CTX_set_verify

Return: none No return.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
void* userCtx = NULL; // Assign some user defined context
...
if(ctx != NULL){
    wolfSSL_SetCertCbCtx(ctx, userCtx);
} else {
```

```
    // Error case, the SSL is not initialized properly.
}



int wolfSSL_CTX_SetChainVerifyCb(
    WOLFSSL_CTX * ctx,
    ChainVerifyCb cb
)
```

Replaces wolfSSL's verification of the peer's certificate chain with an application callback, for every SSL/TLS object created from the context. When a callback is set, wolfSSL decodes the certificates from the Certificate message and hands them to the callback as raw DER, the peer's own certificate first. It builds no chain, verifies no signature and checks no date, revocation status, key usage or host name, and the verify callback set with wolfSSL_CTX_set_verify() is not called; the callback is consulted for any certificate the peer sends, even under WOLFSSL_VERIFY_NONE. A server only requests a client certificate under WOLFSSL_VERIFY_PEER and only requires one with WOLFSSL_VERIFY_FAIL_IF_NO_PEER_CERT, so a server that relies on the callback for client authentication must set both. Malformed DER, a certificate the parser refuses regardless of the verify mode such as one with a zero serial number, or one whose decoding stops part way such as one with an unsupported key algorithm, still fails the handshake before the callback is called. An unknown critical extension or a key usage inconsistent with the basic constraints is left for the callback to judge. The callback returns 0 to accept, CHAIN_VERIFY_WANT_E to suspend the handshake until the application re_enters wolfSSL_connect(), wolfSSL_accept(), wolfSSL_read() or wolfSSL_write() (a certificate received after the handshake is resumed by wolfSSL_read() only, and wolfSSL_write() fails with CHAIN_VERIFY_WANT_E until then), or any other value to reject with CHAIN_VERIFY_CB_E and a fatal bad_certificate alert; 0 is the only accepting value, so returning WOLFSSL_SUCCESS rejects. Because wolfSSL checks nothing itself, wolfSSL_get_verify_result() reports WOLFSSL_X509_V_OK once the callback accepts, recording its verdict rather than a verification wolfSSL performed. DTLS, raw public keys and verifying a stapled OCSP response are not supported with the callback: setting it on a context configured for one of them fails, and so does the handshake of a connection using one of them, with CHAIN_VERIFY_UNSUPPORTED_E. The stapling check covers the client, the side that verifies a stapled response; a server may still staple its own status. Requires WOLFSSL_CHAIN_VERIFY_CB (–enable_chain_verify_cb).

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **cb** the callback, or NULL to clear it.

See:

- wolfSSL_SetChainVerifyCb
- wolfSSL_CTX_SetChainVerifyCtx
- wolfSSL_SetChainVerifyCtx
- wolfSSL_GetChainVerifyCtx

Return:

- WOLFSSL_SUCCESS on success.
- BAD_FUNC_ARG when ctx is NULL.
- CHAIN_VERIFY_UNSUPPORTED_E when the context uses a DTLS method, raw public keys or client-side OCSP stapling.

Example

```
static int myChainVerify(WOLFSSL* ssl, const WOLFSSL_BUFFER_INFO* certs,
                         int certsSz, void* ctx)
{
    // certs[0] is the peer's certificate, the rest is the chain it sent
    if (!hsmVerifyStarted(certs, certsSz))
        return CHAIN_VERIFY_WANT_E; // ask again later
    return hsmVerifyPassed() ? 0 : -1;
}
...
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(method);
if (wolfSSL_CTX_SetChainVerifyCb(ctx, myChainVerify) != WOLFSSL_SUCCESS) {
    // context uses DTLS, raw public keys or client-side stapling
}
```

```
int wolfSSL_SetChainVerifyCb(
    WOLFSSL * ssl,
    ChainVerifyCb cb
)
```

Sets the chain verification callback for one SSL/TLS object. It takes precedence over the callback set on the context with wolfSSL_CTX_SetChainVerifyCb(), which documents the callback's contract.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **cb** the callback, or NULL to fall back to the context's.

See:

- wolfSSL_CTX_SetChainVerifyCb
- wolfSSL_SetChainVerifyCtx

Return:

- WOLFSSL_SUCCESS on success.
- BAD_FUNC_ARG when ssl is NULL.
- CHAIN_VERIFY_UNSUPPORTED_E when the object uses DTLS, raw public keys or client-side OCSP stapling.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
if (wolfSSL_SetChainVerifyCb(ssl, myChainVerify) != WOLFSSL_SUCCESS) {
    // object uses DTLS, raw public keys or client-side stapling
}
```

```
void wolfSSL_CTX_SetChainVerifyCtx(
    WOLFSSL_CTX * ctx,
    void * userCtx
)
```

Sets the user context passed to the chain verification callback for every SSL/TLS object created from the context.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **userCtx** the value the callback receives as its ctx argument.

See:

- wolfSSL_CTX_SetChainVerifyCb
- wolfSSL_SetChainVerifyCtx
- wolfSSL_GetChainVerifyCtx

Return: none No return.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(method);
wolfSSL_CTX_SetChainVerifyCb(ctx, myChainVerify);
wolfSSL_CTX_SetChainVerifyCtx(ctx, &myHsm);
```

```
void wolfSSL_SetChainVerifyCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

Sets the user context passed to the chain verification callback for one SSL/TLS object. It takes precedence over the value set on the context with wolfSSL_CTX_SetChainVerifyCtx().

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **ctx** the value the callback receives as its ctx argument, or NULL to fall back to the context's.

See:

- wolfSSL_CTX_SetChainVerifyCtx
- wolfSSL_GetChainVerifyCtx

Return: none No return.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
wolfSSL_SetChainVerifyCtx(ssl, &myConnectionState);
```

```
void * wolfSSL_GetChainVerifyCtx(
    const WOLFSSL * ssl
)
```

Returns the user context the chain verification callback is called with for this SSL/TLS object: the value set with wolfSSL_SetChainVerifyCtx() when there is one, otherwise the context's.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See:

- wolfSSL_SetChainVerifyCtx
- wolfSSL_CTX_SetChainVerifyCtx

Return: void* the user context, or NULL when none is set or ssl is NULL.

Example

```
struct myState* st = (struct myState*)wolfSSL_GetChainVerifyCtx(ssl);
```

```
int wolfSSL_CTX_save_cert_cache(
    WOLFSSL_CTX * ctx,
    const char * fname
)
```

This function writes the cert cache from memory to file.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, holding the certificate information.
- **fname** a constant char pointer that points to a file for writing.

See:

- CM_SaveCertCache
- DoMemSaveCertCache

Return:

- SSL_SUCCESS if CM_SaveCertCache exits normally.
- BAD_FUNC_ARG is returned if either of the arguments are NULL.
- SSL_BAD_FILE if the cert cache save file could not be opened.
- BAD_MUTEX_E if the lock mutex failed.

- MEMORY_E the allocation of memory failed.
- FWRITE_ERROR Certificate cache file write failed.

Example

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new( protocol def );
const char* fname;
...
if(wolfSSL_CTX_save_cert_cache(ctx, fname)){
    // file was written.
}
```

```
int wolfSSL_CTX_restore_cert_cache(
    WOLFSSL_CTX * ctx,
    const char * fname
)
```

This function persistes certificate cache from a file.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, holding the certificate information.
- **fname** a constant char pointer that points to a file for reading.

See:

- CM_RestoreCertCache
- XFOPEN

Return:

- SSL_SUCCESS returned if the function, CM_RestoreCertCache, executes normally.
- SSL_BAD_FILE returned if XFOPEN returns XBADFILE. The file is corrupted.
- MEMORY_E returned if the allocated memory for the temp buffer fails.
- BAD_FUNC_ARG returned if fname or ctx have a NULL value.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
const char* fname = "path to file";
...
if(wolfSSL_CTX_restore_cert_cache(ctx, fname)){
    // check to see if the execution was successful
}
```

```
int wolfSSL_CTX_memsave_cert_cache(
    WOLFSSL_CTX * ctx,
    void * mem,
    int sz,
    int * used
)
```

This function persists the certificate cache to memory.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **mem** a void pointer to the destination (output buffer).
- **sz** the size of the output buffer.
- **used** a pointer to size of the cert cache header.

See:

- DoMemSaveCertCache
- GetCertCacheMemSize
- CM_MemRestoreCertCache
- CM_GetCertCacheMemSize

Return:

- SSL_SUCCESS returned on successful execution of the function. No errors were thrown.
- BAD_MUTEX_E mutex error where the WOLFSSL_CERT_MANAGER member caLock was not 0 (zero).
- BAD_FUNC_ARG returned if ctx, mem, or used is NULL or if sz is less than or equal to 0 (zero).
- BUFFER_E output buffer mem was too small.

Example

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new( protocol );
void* mem;
int sz;
int* used;
...
if(wolfSSL_CTX_memsave_cert_cache(ctx, mem, sz, used) != SSL_SUCCESS){
    // The function returned with an error
}
```

```
int wolfSSL_CTX_get_cert_cache_memsize(
    WOLFSSL_CTX * ctx
)
```

Returns the size the certificate cache save buffer needs to be.

Parameters:

- **ctx** a pointer to a wolfSSL_CTX structure, created using wolfSSL_CTX_new().

See: CM_GetCertCacheMemSize

Return:

- int integer value returned representing the memory size upon success.
- BAD_FUNC_ARG is returned if the WOLFSSL_CTX struct is NULL.
- BAD_MUTEX_E - returned if there was a mutex lock error.

Example

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(protocol);
...
int certCacheSize = wolfSSL_CTX_get_cert_cache_memsize(ctx);

if(certCacheSize != BAD_FUNC_ARG || certCacheSize != BAD_MUTEX_E){
// Successfully retrieved the memory size.
}
```

```
char * wolfSSL_X509_NAME_oneline(
    WOLFSSL_X509_NAME * name,
    char * in,
    int sz
)
```

This function copies the name of the x509 into a buffer.

Parameters:

- **name** a pointer to a WOLFSSL_X509 structure.
- **in** a buffer to hold the name copied from the WOLFSSL_X509_NAME structure.
- **sz** the maximum size of the buffer.

See:

- wolfSSL_X509_get_subject_name
- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_isCA
- wolfSSL_get_peer_certificate
- wolfSSL_X509_version

Return: A char pointer to the buffer with the WOLFSSL_X509_NAME structures name member's data is returned if the function executed normally.

Example

```
WOLFSSL_X509 x509;
char* name;
...
name = wolfSSL_X509_NAME_oneline(wolfSSL_X509_get_issuer_name(x509), 0, 0);
```

```
if(name <= 0){
    // There's nothing in the buffer.
}
```

```
WOLFSSL_X509_NAME * wolfSSL_X509_get_issuer_name(
    WOLFSSL_X509 * cert
)
```

This function returns the name of the certificate issuer.

Parameters:

- **cert** a pointer to a WOLFSSL_X509 structure.

See:

- wolfSSL_X509_get_subject_name
- wolfSSL_X509_get_isCA
- wolfSSL_get_peer_certificate
- wolfSSL_X509_NAME_oneline

Return:

- point a pointer to the WOLFSSL_X509 struct's issuer member is returned.
- NULL if the cert passed in is NULL.

Example

```
WOLFSSL_X509* x509;
WOLFSSL_X509_NAME issuer;
...
issuer = wolfSSL_X509_NAME_oneline(wolfSSL_X509_get_issuer_name(x509), 0, 0);

if(!issuer){
    // NULL was returned
} else {
    // issuer hods the name of the certificate issuer.
}
```

```
WOLFSSL_X509_NAME * wolfSSL_X509_get_subject_name(
    WOLFSSL_X509 * cert
)
```

This function returns the subject member of the WOLFSSL_X509 structure.

Parameters:

- **cert** a pointer to a WOLFSSL_X509 structure.

See:

- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_isCA
- wolfSSL_get_peer_certificate

Return: pointer a pointer to the WOLFSSL_X509_NAME structure. The pointer may be NULL if the WOLFSSL_X509 struct is NULL or if the subject member of the structure is NULL.

Example

```
WOLFSSL_X509* cert;
WOLFSSL_X509_NAME name;
…
name = wolfSSL_X509_get_subject_name(cert);
if(name == NULL){
    // Deal with the NULL cacse
}
```

```
int wolfSSL_X509_get_isCA(
    WOLFSSL_X509 * x509
)
```

Checks the isCa member of the WOLFSSL_X509 structure and returns the value.

Parameters:

- **x509** a pointer to a WOLFSSL_X509 structure.

See:

- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_isCA

Return:

- isCA returns the value in the isCA member of the WOLFSSL_X509 structure is returned.
- 0 returned if there is not a valid x509 structure passed in.

Example

```
WOLFSSL* ssl;
...
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_X509_get_isCA(ssl)){
    // This is the CA
```

```
}else {
    // Failure case
}
```

```
int wolfSSL_X509_NAME_get_text_by_NID(
    const WOLFSSL_X509_NAME * name,
    int nid,
    char * buf,
    int len
)
```

This function gets the text related to the passed in NID value.

Parameters:

- **name** WOLFSSL_X509_NAME to search for text.
- **nid** NID to search for.
- **buf** buffer to hold text when found.
- **len** length of buffer.

See: none

Return: int returns the size of the text buffer.

Example

```
WOLFSSL_X509_NAME* name;
char buffer[100];
int bufferSz;
int ret;
// get WOLFSSL_X509_NAME
ret = wolfSSL_X509_NAME_get_text_by_NID(name, NID_commonName,
buffer, bufferSz);

//check ret value
```

```
int wolfSSL_X509_get_signature_type(
    WOLFSSL_X509 * x509
)
```

This function returns the value stored in the sigOID member of the WOLFSSL_X509 structure.

Parameters:

- **x509** a pointer to a WOLFSSL_X509 structure.

See:

- wolfSSL_X509_get_signature

- wolfSSL_X509_version
- wolfSSL_X509_get_der
- wolfSSL_X509_get_serial_number
- wolfSSL_X509_notBefore
- wolfSSL_X509_notAfter
- wolfSSL_X509_free

Return:

- 0 returned if the WOLFSSL_X509 structure is NULL.
- int an integer value is returned which was retrieved from the x509 object.

Example

```
WOLFSSL_X509 x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                        DYNAMIC_TYPE_X509);
...
int x509SigType = wolfSSL_X509_get_signature_type(x509);

if(x509SigType != EXPECTED){
// Deal with an unexpected value
}
```

```
int wolfSSL_X509_get_signature(
    WOLFSSL_X509 * x509,
    unsigned char * buf,
    int * bufSz
)
```

Gets the X509 signature and stores it in the buffer.

Parameters:

- **x509** pointer to a WOLFSSL_X509 structure.
- **buf** a char pointer to the buffer.
- **bufSz** an integer pointer to the size of the buffer.

See:

- wolfSSL_X509_get_serial_number
- wolfSSL_X509_get_signature_type
- wolfSSL_X509_get_device_type

Return:

- SSL_SUCCESS returned if the function successfully executes. The signature is loaded into the buffer.
- SSL_FATAL_ERRROR returns if the x509 struct or the bufSz member is NULL. There is also a check for the length member of the sig structure (sig is a member of x509).

Example

```
WOLFSSL_X509* x509 = (WOLFSSL_X509)XMALOC(sizeof(WOLFSSL_X509), NULL,
DYNAMIC_TYPE_X509);
unsigned char* buf; // Initialize
int* bufSz = sizeof(buf)/sizeof(unsigned char);
...
if(wolfSSL_X509_get_signature(x509, buf, bufSz) != SSL_SUCCESS){
    // The function did not execute successfully.
} else{
    // The buffer was written to correctly.
}
```

```
int wolfSSL_X509_STORE_add_cert(
    WOLFSSL_X509_STORE * store,
    WOLFSSL_X509 * x509
)
```

This function adds a certificate to the WOLFSSL_X509_STRE structure.

Parameters:

- **store** certificate store to add the certificate to.
- **x509** certificate to add.

See: wolfSSL_X509_free

Return:

- SSL_SUCCESS If certificate is added successfully.
- SSL_FATAL_ERROR: If certificate is not added successfully.

Example

```
WOLFSSL_X509_STORE* str;
WOLFSSL_X509* x509;
int ret;
ret = wolfSSL_X509_STORE_add_cert(str, x509);
//check ret value
```

```
WOLFSSL_STACK * wolfSSL_X509_STORE_CTX_get_chain(
    WOLFSSL_X509_STORE_CTX * ctx
)
```

This function is a getter function for chain variable in WOLFSSL_X509_STORE_CTX structure. Currently chain is not populated.

Parameters:

- **ctx** certificate store ctx to get parse chain from.

See: wolfSSL_sk_X509_free

Return:

- pointer if successful returns WOLFSSL_STACK (same as STACK_OF(WOLFSSL_X509)) pointer
- Null upon failure

Example

```
WOLFSSL_STACK* sk;
WOLFSSL_X509_STORE_CTX* ctx;
sk = wolfSSL_X509_STORE_CTX_get_chain(ctx);
//check sk for NULL and then use it. sk needs freed after done.
```

```
int wolfSSL_X509_STORE_set_flags(
    WOLFSSL_X509_STORE * store,
    unsigned long flag
)
```

This function takes in a flag to change the behavior of the WOLFSSL_X509_STORE structure passed in. An example of a flag used is WOLFSSL_CRL_CHECK. X509_V_FLAG_CRL_CHECK and X509_V_FLAG_CRL_CHECK_ALL enable CRL checking like WOLFSSL_CRL_CHECK and WOLFSSL_CRL_CHECKALL.

Parameters:

- **str** certificate store to set flag in.
- **flag** flag for behavior.

See:

- wolfSSL_X509_STORE_new
- wolfSSL_X509_STORE_free

Return:

- SSL_SUCCESS If no errors were encountered when setting the flag.
- <0 a negative value will be returned upon failure.

Example

```
WOLFSSL_X509_STORE* str;
int ret;
// create and set up str
ret = wolfSSL_X509_STORE_set_flags(str, WOLFSSL_CRL_CHECKALL);
If (ret != SSL_SUCCESS) {
    //check ret value and handle error case
}
```

```
const byte * wolfSSL_X509_notBefore(
    WOLFSSL_X509 * x509
)
```

This function the certificate “not before” validity encoded as a byte array.

Parameters:

- **x509** pointer to a WOLFSSL_X509 structure.

See:

- wolfSSL_X509_get_signature
- wolfSSL_X509_version
- wolfSSL_X509_get_der
- wolfSSL_X509_get_serial_number
- wolfSSL_X509_notAfter
- wolfSSL_X509_free

Return:

- NULL returned if the WOLFSSL_X509 structure is NULL.
- byte is returned that contains the notBeforeData.

Example

```
WOLFSSL_X509* x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                        DYNAMIC_TYPE_X509);
...
byte* notBeforeData = wolfSSL_X509_notBefore(x509);
```

```
const byte * wolfSSL_X509_notAfter(
    WOLFSSL_X509 * x509
)
```

This function the certificate “not after” validity encoded as a byte array.

Parameters:

- **x509** pointer to a WOLFSSL_X509 structure.

See:

- wolfSSL_X509_get_signature
- wolfSSL_X509_version
- wolfSSL_X509_get_der
- wolfSSL_X509_get_serial_number
- wolfSSL_X509_notBefore
- wolfSSL_X509_free

Return:

- NULL returned if the WOLFSSL_X509 structure is NULL.
- byte is returned that contains the notAfterData.

Example

```
WOLFSSL_X509* x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                        DYNAMIC_TYPE_X509);
...
byte* notAfterData = wolfSSL_X509_notAfter(x509);
```

```
void wolfSSL_CTX_set_client_cert_cb(
    WOLFSSL_CTX * ctx,
    client_cert_cb cb
)
```

Sets a callback to select the client certificate and private key.

Parameters:

- **ctx** The WOLFSSL_CTX object.
- **cb** The callback function to select the client certificate and key.

See: wolfSSL_CTX_set_cert_cb

Return: void

This function allows the application to register a callback that will be invoked when a client certificate is requested during the handshake. The callback can select and provide the certificate and key to use.

Example

```
int my_client_cert_cb(WOLFSSL *ssl, WOLFSSL_X509 **x509, WOLFSSL_EVP_PKEY
 ↪  **pkey) { ... }
wolfSSL_CTX_set_client_cert_cb(ctx, my_client_cert_cb);
```

```
void wolfSSL_CTX_set_cert_cb(
    WOLFSSL_CTX * ctx,
    CertSetupCallback cb,
    void * arg
)
```

Sets a generic certificate setup callback.

Parameters:

- **ctx** The WOLFSSL_CTX object.
- **cb** The callback function for certificate setup.

- **arg** User argument to pass to the callback.

See:

- wolfSSL_CTX_set_client_cert_cb
- wolfSSL_get0_peer_CA_list
- wolfSSL_get_client_CA_list

Return: void

The callback is called whenever a certificate is about to be used, so the application can inspect, set or clear certificates - for example to react to a CA list sent by the peer.

Example

```
int my_cert_setup_cb(WOLFSSL* ssl, void* arg) { ... }
wolfSSL_CTX_set_cert_cb(ctx, my_cert_setup_cb, NULL);
```

```
const char * wolfSSL_get_psk_identity_hint(
    const WOLFSSL *
)
```

This function returns the psk identity hint.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_get_psk_identity

Return:

- pointer a const char pointer to the value that was stored in the arrays member of the WOLFSSL structure is returned.
- NULL returned if the WOLFSSL or Arrays structures are NULL.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
char* idHint;
...
idHint = wolfSSL_get_psk_identity_hint(ssl);
if(idHint){
    // The hint was retrieved
    return idHint;
} else {
    // Hint wasn't successfully retrieved
}
```

```
const char * wolfSSL_get_psk_identity(
    const WOLFSSL *
)
```

The function returns a constant pointer to the client_identity member of the Arrays structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_get_psk_identity_hint
- wolfSSL_use_psk_identity_hint

Return:

- string the string value of the client_identity member of the Arrays structure.
- NULL if the WOLFSSL structure is NULL or if the Arrays member of the WOLFSSL structure is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
const char* pskID;
...
pskID = wolfSSL_get_psk_identity(ssl);

if(pskID == NULL){
    // There is not a value in pskID
}
```

```
int wolfSSL_CTX_use_psk_identity_hint(
    WOLFSSL_CTX * ctx,
    const char * hint
)
```

This function stores the hint argument in the server_hint member of the WOLFSSL_CTX structure.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **hint** a constant char pointer that will be copied to the WOLFSSL_CTX structure.

See: wolfSSL_use_psk_identity_hint

Return: SSL_SUCCESS returned for successful execution of the function.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
const char* hint;
int ret;
…
ret = wolfSSL_CTX_use_psk_identity_hint(ctx, hint);
if(ret == SSL_SUCCESS){
    // Function was successful.
return ret;
} else {
    // Failure case.
}
```

```
int wolfSSL_use_psk_identity_hint(
    WOLFSSL * ssl,
    const char * hint
)
```

This function stores the hint argument in the server_hint member of the Arrays structure within the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **hint** a constant character pointer that holds the hint to be saved in memory.

See: wolfSSL_CTX_use_psk_identity_hint

Return:

- SSL_SUCCESS returned if the hint was successfully stored in the WOLFSSL structure.
- SSL_FAILURE returned if the WOLFSSL or Arrays structures are NULL.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
const char* hint;
...
if(wolfSSL_use_psk_identity_hint(ssl, hint) != SSL_SUCCESS){
    // Handle failure case.
}
```

```
WOLFSSL_X509 * wolfSSL_get_peer_certificate(
    WOLFSSL * ssl
)
```

This function gets the peer's certificate.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_subject_name
- wolfSSL_X509_get_isCA

Return:

- pointer a pointer to the peerCert member of the WOLFSSL_X509 structure if it exists.
- 0 returned if the peer certificate issuer size is not defined.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
WOLFSSL_X509* peerCert = wolfSSL_get_peer_certificate(ssl);

if(peerCert){
    // You have a pointer peerCert to the peer certification
}
```

```
WOLFSSL_X509 * wolfSSL_get_chain_X509(
    WOLFSSL_X509_CHAIN * chain,
    int idx
)
```

This function gets the peer's wolfSSL_X509_certificate at index (idx) from the chain of certificates.

Parameters:

- **chain** a pointer to the WOLFSSL_X509_CHAIN used for no dynamic memory SESSION_CACHE.
- **idx** the index of the WOLFSSL_X509 certificate.

See:

- InitDecodedCert
- ParseCertRelative
- CopyDecodedToX509

Return: pointer returns a pointer to a WOLFSSL_X509 structure.

Note that it is the user's responsibility to free the returned memory by calling wolfSSL_FreeX509(). NULL is also returned when the slot at idx holds no certificate because the one presented there was too large to store, which can happen for an index inside wolfSSL_get_chain_count().

Example

```
WOLFSSL_X509_CHAIN* chain = &session->chain;
int idx = 999; // set idx
...
WOLFSSL_X509_CHAIN ptr;
prt = wolfSSL_get_chain_X509(chain, idx);

if(ptr != NULL){
    // ptr contains the cert at the index specified
    wolfSSL_FreeX509(ptr);
} else {
    // ptr is NULL
}
```

```
char * wolfSSL_X509_get_subjectCN(
    WOLFSSL_X509 *
)
```

Returns the common name of the subject from the certificate.

Parameters:

- **x509** a pointer to a WOLFSSL_X509 structure containing certificate information.

See:

- wolfSSL_X509_Name_get_entry
- wolfSSL_X509_get_next_altname
- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_subject_name

Return:

- NULL returned if the x509 structure is null
- string a string representation of the subject's common name is returned upon success

Example

```
WOLFSSL_X509 x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                        DYNAMIC_TYPE_X509);
...
int x509Cn = wolfSSL_X509_get_subjectCN(x509);
if(x509Cn == NULL){
    // Deal with NULL case
} else {
    // x509Cn contains the common name
}
```

```
const unsigned char * wolfSSL_X509_get_der(
    WOLFSSL_X509 * x509,
    int * outSz
)
```

This function gets the DER encoded certificate in the WOLFSSL_X509 struct.

Parameters:

- **x509** a pointer to a WOLFSSL_X509 structure containing certificate information.
- **outSz** length of the derBuffer member of the WOLFSSL_X509 struct.

See:

- wolfSSL_X509_version
- wolfSSL_X509_Name_get_entry
- wolfSSL_X509_get_next_altname
- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_subject_name

Return:

- buffer This function returns the DerBuffer structure's buffer member, which is of type byte.
- NULL returned if the x509 or outSz parameter is NULL.

Example

```
WOLFSSL_X509 x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                        DYNAMIC_TYPE_X509);
int* outSz; // initialize
...
byte* x509Der = wolfSSL_X509_get_der(x509, outSz);
if(x509Der == NULL){
    // Failure case one of the parameters was NULL
}
```

```
WOLFSSL_ASN1_TIME * wolfSSL_X509_get_notAfter(
    WOLFSSL_X509 *
)
```

This function checks to see if x509 is NULL and if it's not, it returns the notAfter member of the x509 struct.

Parameters:

- **x509** a pointer to the WOLFSSL_X509 struct.

See: wolfSSL_X509_get_notBefore

Return:

- pointer to struct with ASN1_TIME to the notAfter member of the x509 struct.
- NULL returned if the x509 object is NULL.

Example

```
WOLFSSL_X509* x509 = (WOLFSSL_X509)XMALOC(sizeof(WOLFSSL_X509), NULL,
DYNAMIC_TYPE_X509) ;
...
const WOLFSSL_ASN1_TIME* notAfter = wolfSSL_X509_get_notAfter(x509);
if(notAfter == NULL){
    // Failure case, the x509 object is null.
}
```

```
int wolfSSL_X509_version(
    WOLFSSL_X509 * x509
)
```

This function retrieves the version of the X509 certificate.

Parameters:

- **x509** a pointer to a WOLFSSL_X509 structure.

See:

- wolfSSL_X509_get_subject_name
- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_isCA
- wolfSSL_get_peer_certificate

Return:

- 0 returned if the x509 structure is NULL.
- version the version stored in the x509 structure will be returned.

Example

```
WOLFSSL_X509* x509;
int version;
...
version = wolfSSL_X509_version(x509);
if(!version){
    // The function returned 0, failure case.
}
```

```
WOLFSSL_X509 * wolfSSL_X509_d2i_fp(
    WOLFSSL_X509 ** x509,
    FILE * file
)
```

If NO_STDIO_FILESYSTEM is defined this function will allocate heap memory, initialize a WOLFSSL_X509 structure and return a pointer to it.

Parameters:

- **x509** a pointer to a WOLFSSL_X509 pointer.
- **file** a defined type that is a pointer to a FILE.

See:

- wolfSSL_X509_d2i
- XFTELL
- XREWIND
- XFSEEK

Return:

- *WOLFSSL_X509 WOLFSSL_X509 structure pointer is returned if the function executes successfully.
- NULL if the call to XFTELL macro returns a negative value.

Example

```
WOLFSSL_X509* x509a = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
DYNAMIC_TYPE_X509);
WOLFSSL_X509** x509 = x509a;
XFILE file;  (mapped to struct fs_file*)
...
WOLFSSL_X509* newX509 = wolfSSL_X509_d2i_fp(x509, file);
if(newX509 == NULL){
    // The function returned NULL
}
```

```
WOLFSSL_X509 * wolfSSL_X509_load_certificate_file(
    const char * fname,
    int format
)
```

The function loads the x509 certificate into memory.

Parameters:

- **fname** the certificate file to be loaded.
- **format** the format of the certificate.

See:

- InitDecodedCert
- PemToDer
- wolfSSL_get_certificate
- AssertNotNull

Return:

- pointer a successful execution returns pointer to a WOLFSSL_X509 structure.
- NULL returned if the certificate was not able to be written.

Example

```
#define cliCert    "certs/client-cert.pem"
…
X509* x509;
…
x509 = wolfSSL_X509_load_certificate_file(cliCert, SSL_FILETYPE_PEM);
AssertNotNull(x509);
```

```
unsigned char * wolfSSL_X509_get_device_type(
    WOLFSSL_X509 * x509,
    unsigned char * in,
    int * inOutSz
)
```

This function copies the device type from the x509 structure to the buffer.

Parameters:

- **x509** pointer to a WOLFSSL_X509 structure, created with WOLFSSL_X509_new().
- **in** a pointer to a byte type that will hold the device type (the buffer).
- **inOutSz** the minimum of either the parameter inOutSz or the deviceTypeSz member of the x509 structure.

See:

- wolfSSL_X509_get_hw_type
- wolfSSL_X509_get_hw_serial_number
- wolfSSL_X509_d2i

Return:

- pointer returns a byte pointer holding the device type from the x509 structure.
- NULL returned if the buffer size is NULL.

Example

```
WOLFSSL_X509* x509 = (WOLFSSL_X509)XMALOC(sizeof(WOLFSSL_X509), NULL,
DYNAMIC_TYPE_X509);
byte* in;
int* inOutSz;
...
byte* deviceType = wolfSSL_X509_get_device_type(x509, in, inOutSz);

if(!deviceType){
    // Failure case, NULL was returned.
}
```

```
unsigned char * wolfSSL_X509_get_hw_type(
    WOLFSSL_X509 * x509,
    unsigned char * in,
    int * inOutSz
)
```

The function copies the hwType member of the WOLFSSL_X509 structure to the buffer.

Parameters:

- **x509** a pointer to a WOLFSSL_X509 structure containing certificate information.
- **in** pointer to type byte that represents the buffer.
- **inOutSz** pointer to type int that represents the size of the buffer.

See:

- wolfSSL_X509_get_hw_serial_number
- wolfSSL_X509_get_device_type

Return:

- byte The function returns a byte type of the data previously held in the hwType member of the WOLFSSL_X509 structure.
- NULL returned if inOutSz is NULL.

Example

```
WOLFSSL_X509* x509;  // X509 certificate
byte* in;  // initialize the buffer
int* inOutSz;  // holds the size of the buffer
...
byte* hwType = wolfSSL_X509_get_hw_type(x509, in, inOutSz);

if(hwType == NULL){
    // Failure case function returned NULL.
}
```

```
unsigned char * wolfSSL_X509_get_hw_serial_number(
    WOLFSSL_X509 * x509,
    unsigned char * in,
    int * inOutSz
)
```

This function returns the hwSerialNum member of the x509 object.

Parameters:

- **x509** pointer to a WOLFSSL_X509 structure containing certificate information.
- **in** a pointer to the buffer that will be copied to.
- **inOutSz** a pointer to the size of the buffer.

See:

- wolfSSL_X509_get_subject_name
- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_isCA
- wolfSSL_get_peer_certificate
- wolfSSL_X509_version

Return: pointer the function returns a byte pointer to the in buffer that will contain the serial number loaded from the x509 object.

Example

```
char* serial;
byte* in;
int* inOutSz;
WOLFSSL_X509 x509;
...
serial = wolfSSL_X509_get_hw_serial_number(x509, in, inOutSz);

if(serial == NULL || serial <= 0){
    // Failure case
}
```

```
int wolfSSL_SetTmpDH(
    WOLFSSL * ssl,
    const unsigned char * p,
    int pSz,
    const unsigned char * g,
    int gSz
)
```

Server Diffie-Hellman Ephemeral parameters setting. This function sets up the group parameters to be used if the server negotiates a cipher suite that uses DHE.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **p** Diffie-Hellman prime number parameter.
- **pSz** size of p.
- **g** Diffie-Hellman "generator" parameter.
- **gSz** size of g.

See: SSL_accept

Return:

- SSL_SUCCESS upon success.
- MEMORY_ERROR will be returned if a memory error was encountered.
- SIDE_ERROR will be returned if this function is called on an SSL client instead of an SSL server.

Example

```
WOLFSSL* ssl;
static unsigned char p[] = {...};
static unsigned char g[] = {...};
...
wolfSSL_SetTmpDH(ssl, p, sizeof(p), g, sizeof(g));
```

```
int wolfSSL_SetTmpDH_buffer(
    WOLFSSL * ssl,
    const unsigned char * b,
    long sz,
    int format
)
```

The function calls the wolfSSL_SetTMpDH_buffer_wrapper, which is a wrapper for Diffie-Hellman parameters.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **buf** allocated buffer passed in from wolfSSL_SetTMpDH_file_wrapper.
- **sz** a long int that holds the size of the file (fname within wolfSSL_SetTmpDH_file_wrapper).
- **format** an integer type passed through from wolfSSL_SetTmpDH_file_wrapper() that is a representation of the certificate format.

See:

- wolfSSL_SetTmpDH_buffer_wrapper
- wc_DhParamsLoad
- wolfSSL_SetTmpDH
- PemToDer
- wolfSSL_CTX_SetTmpDH
- wolfSSL_CTX_SetTmpDH_file

Return:

- SSL_SUCCESS on successful execution.
- SSL_BAD_FILETYPE if the file type is not PEM and is not ASN.1. It will also be returned if the wc_DhParamsLoad does not return normally.
- SSL_NO_PEM_HEADER returns from PemToDer if there is not a PEM header.
- SSL_BAD_FILE returned if there is a file error in PemToDer.
- SSL_FATAL_ERROR returned from PemToDer if there was a copy error.
- MEMORY_E - if there was a memory allocation error.
- BAD_FUNC_ARG returned if the WOLFSSL struct is NULL or if there was otherwise a NULL argument passed to a subroutine.
- DH_KEY_SIZE_E is returned if their is a key size error in wolfSSL_SetTmpDH() or in wolfSSL_CTX_SetTmpDH().
- SIDE_ERROR returned if it is not the server side in wolfSSL_SetTmpDH.

Example

```
Static int wolfSSL_SetTmpDH_file_wrapper(WOLFSSL_CTX* ctx, WOLFSSL* ssl,
Const char* fname, int format);
long sz = 0;
byte* myBuffer = staticBuffer[FILE_BUFFER_SIZE];
…
if(ssl)
ret = wolfSSL_SetTmpDH_buffer(ssl, myBuffer, sz, format);
```

```
int wolfSSL_SetTmpDH_file(
    WOLFSSL * ssl,
    const char * f,
    int format
)
```

This function calls wolfSSL_SetTmpDH_file_wrapper to set server Diffie-Hellman parameters.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **fname** a constant char pointer holding the certificate.
- **format** an integer type that holds the format of the certification.

See:

- wolfSSL_CTX_SetTmpDH_file
- wolfSSL_SetTmpDH_file_wrapper
- wolfSSL_SetTmpDH_buffer
- wolfSSL_CTX_SetTmpDH_buffer
- wolfSSL_SetTmpDH_buffer_wrapper
- wolfSSL_SetTmpDH
- wolfSSL_CTX_SetTmpDH

Return:

- SSL_SUCCESS returned on successful completion of this function and its subroutines.

- MEMORY_E returned if a memory allocation failed in this function or a subroutine.
- SIDE_ERROR if the side member of the Options structure found in the WOLFSSL struct is not the server side.
- SSL_BAD_FILETYPE returns if the certificate fails a set of checks.
- DH_KEY_SIZE_E returned if the DH parameter's key size is less than the value of the minDhKeySz member in the WOLFSSL struct.
- DH_KEY_SIZE_E returned if the DH parameter's key size is greater than the value of the maxDhKeySz member in the WOLFSSL struct.
- BAD_FUNC_ARG returns if an argument value is NULL that is not permitted such as, the WOLFSSL structure.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
const char* dhParam;
…
AssertIntNE(SSL_SUCCESS,
wolfSSL_SetTmpDH_file(ssl, dhParam, SSL_FILETYPE_PEM));
```

```
int wolfSSL_CTX_SetTmpDH(
    WOLFSSL_CTX * ctx,
    const unsigned char * p,
    int pSz,
    const unsigned char * g,
    int gSz
)
```

Sets the parameters for the server CTX Diffie-Hellman.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **p** a constant unsigned char pointer loaded into the buffer member of the serverDH_P struct.
- **pSz** an int type representing the size of p, initialized to MAX_DH_SIZE.
- **g** a constant unsigned char pointer loaded into the buffer member of the serverDH_G struct.
- **gSz** an int type representing the size of g, initialized to MAX_DH_SIZE.

See:

- wolfSSL_SetTmpDH
- wc_DhParamsLoad

Return:

- SSL_SUCCESS returned if the function and all subroutines return without error.
- BAD_FUNC_ARG returned if the CTX, p or g parameters are NULL.
- DH_KEY_SIZE_E returned if the DH parameter's key size is less than the value of the minDhKeySz member of the WOLFSSL_CTX struct.
- DH_KEY_SIZE_E returned if the DH parameter's key size is greater than the value of the maxDhKeySz member of the WOLFSSL_CTX struct.

- MEMORY_E returned if the allocation of memory failed in this function or a subroutine.

Exmaple

```
WOLFSSL_CTX* ctx =  WOLFSSL_CTX_new( protocol );
byte* p;
byte* g;
word32 pSz = (word32)sizeof(p)/sizeof(byte);
word32 gSz = (word32)sizeof(g)/sizeof(byte);
…
int ret =  wolfSSL_CTX_SetTmpDH(ctx, p, pSz, g, gSz);

if(ret != SSL_SUCCESS){
    // Failure case
}
```

```
int wolfSSL_CTX_SetTmpDH_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * b,
    long sz,
    int format
)
```

A wrapper function that calls wolfSSL_SetTmpDH_buffer_wrapper.

Parameters:

- **ctx** a pointer to a WOLFSSL structure, created using wolfSSL_CTX_new().
- **buf** a pointer to a constant unsigned char type that is allocated as the buffer and passed through to wolfSSL_SetTmpDH_buffer_wrapper.
- **sz** a long integer type that is derived from the fname parameter in wolfSSL_SetTmpDH_file_wrapper().
- **format** an integer type passed through from wolfSSL_SetTmpDH_file_wrapper().

See:

- wolfSSL_SetTmpDH_buffer_wrapper
- wolfSSL_SetTMpDH_buffer
- wolfSSL_SetTmpDH_file_wrapper
- wolfSSL_CTX_SetTmpDH_file

Return:

- 0 returned for a successful execution.
- BAD_FUNC_ARG returned if the ctx or buf parameters are NULL.
- MEMORY_E if there is a memory allocation error.
- SSL_BAD_FILETYPE returned if format is not correct.

Example

```
static int wolfSSL_SetTmpDH_file_wrapper(WOLFSSL_CTX* ctx, WOLFSSL* ssl,
Const char* fname, int format);
#ifdef WOLFSSL_SMALL_STACK
byte staticBuffer[1]; // force heap usage
#else
byte* staticBuffer;
long sz = 0;
…
if(ssl){
    ret = wolfSSL_SetTmpDH_buffer(ssl, myBuffer, sz, format);
} else {
ret = wolfSSL_CTX_SetTmpDH_buffer(ctx, myBuffer, sz, format);
}
```

```
int wolfSSL_CTX_SetTmpDH_file(
    WOLFSSL_CTX * ctx,
    const char * f,
    int format
)
```

The function calls wolfSSL_SetTmpDH_file_wrapper to set the server Diffie-Hellman parameters.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **fname** a constant character pointer to a certificate file.
- **format** an integer type passed through from wolfSSL_SetTmpDH_file_wrapper() that is a representation of the certificate format.

See:

- wolfSSL_SetTmpDH_buffer_wrapper
- wolfSSL_SetTmpDH
- wolfSSL_CTX_SetTmpDH
- wolfSSL_SetTmpDH_buffer
- wolfSSL_CTX_SetTmpDH_buffer
- wolfSSL_SetTmpDH_file_wrapper
- AllocDer
- PemToDer

Return:

- SSL_SUCCESS returned if the wolfSSL_SetTmpDH_file_wrapper or any of its subroutines return successfully.
- MEMORY_E returned if an allocation of dynamic memory fails in a subroutine.
- BAD_FUNC_ARG returned if the ctx or fname parameters are NULL or if a subroutine is passed a NULL argument.
- SSL_BAD_FILE returned if the certificate file is unable to open or if the a set of checks on the file fail from wolfSSL_SetTmpDH_file_wrapper.
- SSL_BAD_FILETYPE returned if the format is not PEM or ASN.1 from wolfSSL_SetTmpDH_buffer_wrapper().

- DH_KEY_SIZE_E returned if the DH parameter's key size is less than the value of the minDhKeySz member of the WOLFSSL_CTX struct.
- DH_KEY_SIZE_E returned if the DH parameter's key size is greater than the value of the maxDhKeySz member of the WOLFSSL_CTX struct.
- SIDE_ERROR returned in wolfSSL_SetTmpDH() if the side is not the server end.
- SSL_NO_PEM_HEADER returned from PemToDer if there is no PEM header.
- SSL_FATAL_ERROR returned from PemToDer if there is a memory copy failure.

Example

```
#define dhParam     "certs/dh2048.pem"
#DEFINE aSSERTiNTne(x, y)     AssertInt(x, y, !=, ==)
WOLFSSL_CTX* ctx;
…
AssertNotNull(ctx = wolfSSL_CTX_new(wolfSSLv23_client_method()))
…
AssertIntNE(SSL_SUCCESS, wolfSSL_CTX_SetTmpDH_file(NULL, dhParam,
SSL_FILETYPE_PEM));
```

```
int wolfSSL_CTX_SetMinDhKey_Sz(
    WOLFSSL_CTX * ctx,
    word16 keySz_bits
)
```

This function sets the minimum size (in bits) of the Diffie Hellman key size by accessing the minDhKeySz member in the WOLFSSL_CTX structure.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **keySz_bits** a word16 type used to set the minimum DH key size in bits. The WOLFSSL_CTX struct holds this information in the minDhKeySz member.

See:

- wolfSSL_SetMinDhKey_Sz
- wolfSSL_CTX_SetMaxDhKey_Sz
- wolfSSL_SetMaxDhKey_Sz
- wolfSSL_GetDhKey_Sz
- wolfSSL_CTX_SetTMpDH_file

Return:

- SSL_SUCCESS returned if the function completes successfully.
- BAD_FUNC_ARG returned if the WOLFSSL_CTX struct is NULL or if the keySz_bits is greater than 16,000 or not divisible by 8.

Example

```
public static int CTX_SetMinDhKey_Sz(IntPtr ctx, short minDhKey){
…
return wolfSSL_CTX_SetMinDhKey_Sz(local_ctx, minDhKeyBits);
```

```
int wolfSSL_SetMinDhKey_Sz(
    WOLFSSL * ssl,
    word16 keySz_bits
)
```

Sets the minimum size (in bits) for a Diffie-Hellman key in the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **keySz_bits** a word16 type used to set the minimum DH key size in bits. The WOLFSSL_CTX struct holds this information in the minDhKeySz member.

See:

- wolfSSL_CTX_SetMinDhKey_Sz
- wolfSSL_GetDhKey_Sz

Return:

- SSL_SUCCESS the minimum size was successfully set.
- BAD_FUNC_ARG the WOLFSSL structure was NULL or if the keySz_bits is greater than 16,000 or not divisible by 8.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
word16 keySz_bits;
...
if(wolfSSL_SetMinDhKey_Sz(ssl, keySz_bits) != SSL_SUCCESS){
    // Failed to set.
}
```

```
int wolfSSL_CTX_SetMaxDhKey_Sz(
    WOLFSSL_CTX * ctx,
    word16 keySz_bits
)
```

This function sets the maximum size (in bits) of the Diffie Hellman key size by accessing the maxDhKeySz member in the WOLFSSL_CTX structure.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

- **keySz_bits** a word16 type used to set the maximum DH key size in bits. The WOLFSSL_CTX struct holds this information in the maxDhKeySz member.

See:

- wolfSSL_SetMinDhKey_Sz
- wolfSSL_CTX_SetMinDhKey_Sz
- wolfSSL_SetMaxDhKey_Sz
- wolfSSL_GetDhKey_Sz
- wolfSSL_CTX_SetTMpDH_file

Return:

- SSL_SUCCESS returned if the function completes successfully.
- BAD_FUNC_ARG returned if the WOLFSSL_CTX struct is NULL or if the keySz_bits is greater than 16,000 or not divisible by 8.

Example

```
public static int CTX_SetMaxDhKey_Sz(IntPtr ctx, short maxDhKey){
…
return wolfSSL_CTX_SetMaxDhKey_Sz(local_ctx, keySz_bits);
```

```
int wolfSSL_SetMaxDhKey_Sz(
    WOLFSSL * ssl,
    word16 keySz_bits
)
```

Sets the maximum size (in bits) for a Diffie-Hellman key in the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **keySz** a word16 type representing the bit size of the maximum DH key.

See:

- wolfSSL_CTX_SetMaxDhKey_Sz
- wolfSSL_GetDhKey_Sz

Return:

- SSL_SUCCESS the maximum size was successfully set.
- BAD_FUNC_ARG the WOLFSSL structure was NULL or the keySz parameter was greater than the allowable size or not divisible by 8.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
word16 keySz;
...
if(wolfSSL_SetMaxDhKey(ssl, keySz) != SSL_SUCCESS){
    // Failed to set.
}
```

```
int wolfSSL_GetDhKey_Sz(
    WOLFSSL * ssl
)
```

Returns the value of dhKeySz (in bits) that is a member of the options structure. This value represents the Diffie-Hellman key size in bytes.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_SetMinDhKey_sz
- wolfSSL_CTX_SetMinDhKey_Sz
- wolfSSL_CTX_SetTmpDH
- wolfSSL_SetTmpDH
- wolfSSL_CTX_SetTmpDH_file

Return:

- dhKeySz returns the value held in ssl->options.dhKeySz which is an integer value representing a size in bits.
- BAD_FUNC_ARG returns if the WOLFSSL struct is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
int dhKeySz;
...
dhKeySz = wolfSSL_GetDhKey_Sz(ssl);

if(dhKeySz == BAD_FUNC_ARG || dhKeySz <= 0){
    // Failure case
} else {
    // dhKeySz holds the size of the key.
}
```

```
int wolfSSL_CTX_SetMinRsaKey_Sz(
    WOLFSSL_CTX * ctx,
    short keySz
)
```

Sets the minimum RSA key size in both the WOLFSSL_CTX structure and the WOLFSSL_CERT_MANAGER structure.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **keySz** a short integer type stored in minRsaKeySz in the ctx structure and the cm structure converted to bytes.

See: wolfSSL_SetMinRsaKey_Sz

Return:

- SSL_SUCCESS returned on successful execution of the function.
- BAD_FUNC_ARG returned if the ctx structure is NULL or the keySz is less than zero or not divisible by 8.

Example

```
WOLFSSL_CTX* ctx = SSL_CTX_new(method);
(void)minDhKeyBits;
ourCert = myoptarg;
…
minDhKeyBits = atoi(myoptarg);
…
if(wolfSSL_CTX_SetMinRsaKey_Sz(ctx, minRsaKeyBits) != SSL_SUCCESS){
…
```

```
int wolfSSL_SetMinRsaKey_Sz(
    WOLFSSL * ssl,
    short keySz
)
```

Sets the minimum allowable key size in bits for RSA located in the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **keySz** a short integer value representing the the minimum key in bits.

See: wolfSSL_CTX_SetMinRsaKey_Sz

Return:

- SSL_SUCCESS the minimum was set successfully.

- BAD_FUNC_ARG returned if the ssl structure is NULL or if the ksySz is less than zero or not divisible by 8.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
short keySz;
…

int isSet =  wolfSSL_SetMinRsaKey_Sz(ssl, keySz);
if(isSet != SSL_SUCCESS){
    Failed to set.
}
```

```
int wolfSSL_CTX_SetMinEccKey_Sz(
    WOLFSSL_CTX * ctx,
    short keySz
)
```

Sets the minimum size in bits for the ECC key in the WOLF_CTX structure and the WOLFSSL_CERT_MANAGER structure.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **keySz** a short integer type that represents the minimum ECC key size in bits.

See: wolfSSL_SetMinEccKey_Sz

Return:

- SSL_SUCCESS returned for a successful execution and the minEccKeySz member is set.
- BAD_FUNC_ARG returned if the WOLFSSL_CTX struct is NULL or if the keySz is negative or not divisible by 8.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
short keySz; // minimum key size
…
if(wolfSSL_CTX_SetMinEccKey(ctx, keySz) != SSL_SUCCESS){
    // Failed to set min key size
}
```

```
int wolfSSL_SetMinEccKey_Sz(
    WOLFSSL * ssl,
    short keySz
)
```

Sets the value of the minEccKeySz member of the options structure. The options struct is a member of the WOLFSSL structure and is accessed through the ssl parameter.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **keySz** value used to set the minimum ECC key size. Sets value in the options structure.

See:

- wolfSSL_CTX_SetMinEccKey_Sz
- wolfSSL_CTX_SetMinRsaKey_Sz
- wolfSSL_SetMinRsaKey_Sz

Return:

- SSL_SUCCESS if the function successfully set the minEccKeySz member of the options structure.
- BAD_FUNC_ARG if the WOLFSSL_CTX structure is NULL or if the key size (keySz) is less than 0 (zero) or not divisible by 8.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx); // New session
short keySz = 999; // should be set to min key size allowable
...
if(wolfSSL_SetMinEccKey_Sz(ssl, keySz) != SSL_SUCCESS){
    // Failure case.
}
```

```
int wolfSSL_make_eap_keys(
    WOLFSSL * ssl,
    void * key,
    unsigned int len,
    const char * label
)
```

This function is used by EAP_TLS and EAP-TTLS to derive keying material from the master secret.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **key** a void pointer variable that will hold the result of the p_hash function.
- **len** an unsigned integer that represents the length of the key variable.
- **label** a constant char pointer that is copied from in wc_PRF().

See:

- wc_PRF
- wc_HmacFinal

- wc_HmacUpdate

Return:

- BUFFER_E returned if the actual size of the buffer exceeds the maximum size allowable.
- MEMORY_E returned if there is an error with memory allocation.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);;
void* key;
unsigned int len;
const char* label;
…
return wolfSSL_make_eap_keys(ssl, key, len, label);
```

```
int wolfSSL_CTX_load_verify_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format
)
```

This function loads a CA certificate buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. More than one CA certificate may be loaded per buffer as long as the format is in PEM. Please see the examples for proper usage.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **in** pointer to the CA certificate buffer.
- **sz** size of the input CA certificate buffer, in.
- **format** format of the buffer certificate, either SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CTX_load_verify_locations
- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_certificate_chain_buffer
- wolfSSL_use_certificate_buffer
- wolfSSL_use_PrivateKey_buffer
- wolfSSL_use_certificate_chain_buffer

Return:

- SSL_SUCCESS upon success
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.

- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- BUFFER_E will be returned if a chain buffer is bigger than the receiving buffer.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
byte certBuff[...];
long sz = sizeof(certBuff);
...

ret = wolfSSL_CTX_load_verify_buffer(ctx, certBuff, sz, SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading CA certs from buffer
}
...
```

```
int wolfSSL_CTX_load_verify_buffer_ex(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format,
    int userChain,
    word32 flags
)
```

This function loads a CA certificate buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. More than one CA certificate may be loaded per buffer as long as the format is in PEM. The _ex version was added in PR 2413 and supports additional arguments for userChain and flags.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **in** pointer to the CA certificate buffer.
- **sz** size of the input CA certificate buffer, in.
- **format** format of the buffer certificate, either SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.
- **userChain** If using format WOLFSSL_FILETYPE_ASN1 this set to non-zero indicates a chain of DER's is being presented.
- **flags** See ssl.h around WOLFSSL_LOAD_VERIFY_DEFAULT_FLAGS.

See:

- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_load_verify_locations
- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_CTX_use_PrivateKey_buffer

- wolfSSL_CTX_use_certificate_chain_buffer
- wolfSSL_use_certificate_buffer
- wolfSSL_use_PrivateKey_buffer
- wolfSSL_use_certificate_chain_buffer

Return:

- SSL_SUCCESS upon success
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- BUFFER_E will be returned if a chain buffer is bigger than the receiving buffer.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
byte certBuff[...];
long sz = sizeof(certBuff);
...

// Example for force loading an expired certificate
ret = wolfSSL_CTX_load_verify_buffer_ex(ctx, certBuff, sz, SSL_FILETYPE_PEM,
    0, (WOLFSSL_LOAD_FLAG_DATE_ERR_OKAY));
if (ret != SSL_SUCCESS) {
    // error loading CA certs from buffer
}
...
```

```
int wolfSSL_CTX_load_verify_chain_buffer_format(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format
)
```

This function loads a CA certificate chain buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. More than one CA certificate may be loaded per buffer as long as the format is in PEM. Please see the examples for proper usage.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **in** pointer to the CA certificate buffer.
- **sz** size of the input CA certificate buffer, in.
- **format** format of the buffer certificate, either SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CTX_load_verify_locations
- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_certificate_chain_buffer
- wolfSSL_use_certificate_buffer
- wolfSSL_use_PrivateKey_buffer
- wolfSSL_use_certificate_chain_buffer

Return:

- SSL_SUCCESS upon success
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- BUFFER_E will be returned if a chain buffer is bigger than the receiving buffer.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
byte certBuff[...];
long sz = sizeof(certBuff);
...

ret = wolfSSL_CTX_load_verify_chain_buffer_format(ctx,
                     certBuff, sz, WOLFSSL_FILETYPE_ASN1);
if (ret != SSL_SUCCESS) {
    // error loading CA certs from buffer
}
...
```

```
int wolfSSL_CTX_use_certificate_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format
)
```

This function loads a certificate buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. Please see the examples for proper usage.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **in** the input buffer containing the certificate to be loaded.
- **sz** the size of the input buffer.

- **format** the format of the certificate located in the input buffer (in). Possible values are SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_certificate_chain_buffer
- wolfSSL_use_certificate_buffer
- wolfSSL_use_PrivateKey_buffer
- wolfSSL_use_certificate_chain_buffer

Return:

- SSL_SUCCESS upon success
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
byte certBuff[...];
long sz = sizeof(certBuff);
...
ret = wolfSSL_CTX_use_certificate_buffer(ctx, certBuff, sz, SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading certificate from buffer
}
...
```

```
int wolfSSL_CTX_use_PrivateKey_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format
)
```

This function loads a private key buffer into the SSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1or SSL_FILETYPE_PEM. Please see the examples for proper usage.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **in** the input buffer containing the private key to be loaded.
- **sz** the size of the input buffer.

- **format** the format of the private key located in the input buffer (in). Possible values are SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_CTX_use_certificate_chain_buffer
- wolfSSL_use_certificate_buffer
- wolfSSL_use_PrivateKey_buffer
- wolfSSL_use_certificate_chain_buffer

Return:

- SSL_SUCCESS upon success
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- NO_PASSWORD will be returned if the key file is encrypted but no password is provided.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
byte keyBuff[...];
long sz = sizeof(certBuff);
...
ret = wolfSSL_CTX_use_PrivateKey_buffer(ctx, keyBuff, sz, SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading private key from buffer
}
...
```

```
int wolfSSL_CTX_use_certificate_chain_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz
)
```

This function loads a certificate chain buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. The buffer must be in PEM format and start with the subject's certificate, ending with the root certificate. Please see the examples for proper usage.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **in** the input buffer containing the PEM-formatted certificate chain to be loaded.
- **sz** the size of the input buffer.

See:

- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_use_certificate_buffer
- wolfSSL_use_PrivateKey_buffer
- wolfSSL_use_certificate_chain_buffer

Return:

- SSL_SUCCESS upon success
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- BUFFER_E will be returned if a chain buffer is bigger than the receiving buffer.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
byte certChainBuff[...];
long sz = sizeof(certBuff);
...
ret = wolfSSL_CTX_use_certificate_chain_buffer(ctx, certChainBuff, sz);
if (ret != SSL_SUCCESS) {
    // error loading certificate chain from buffer
}
...
```

```
int wolfSSL_use_certificate_buffer(
    WOLFSSL * ssl,
    const unsigned char * in,
    long sz,
    int format
)
```

This function loads a certificate buffer into the WOLFSSL object. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. Please see the examples for proper usage.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **in** buffer containing certificate to load.
- **sz** size of the certificate located in buffer.
- **format** format of the certificate to be loaded. Possible values are SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_certificate_chain_buffer
- wolfSSL_use_PrivateKey_buffer
- wolfSSL_use_certificate_chain_buffer

Return:

- SSL_SUCCESS upon success.
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.

Example

```
int ret;
byte certBuff[...];
WOLFSSL* ssl = 0;
long buffSz = sizeof(certBuff);
...

ret = wolfSSL_use_certificate_buffer(ssl, certBuff, buffSz, SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // failed to load certificate from buffer
}
```

```
int wolfSSL_use_PrivateKey_buffer(
    WOLFSSL * ssl,
    const unsigned char * in,
    long sz,
    int format
)
```

This function loads a private key buffer into the WOLFSSL object. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. Please see the examples for proper usage.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **in** buffer containing private key to load.
- **sz** size of the private key located in buffer.
- **format** format of the private key to be loaded. Possible values are SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_use_PrivateKey
- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_certificate_chain_buffer
- wolfSSL_use_certificate_buffer
- wolfSSL_use_certificate_chain_buffer

Return:

- SSL_SUCCESS upon success.
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- NO_PASSWORD will be returned if the key file is encrypted but no password is provided.

Example

```
int ret;
byte keyBuff[...];
WOLFSSL* ssl = 0;
long buffSz = sizeof(certBuff);
...
ret = wolfSSL_use_PrivateKey_buffer(ssl, keyBuff, buffSz, SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // failed to load private key from buffer
}
```

```
int wolfSSL_use_certificate_chain_buffer(
    WOLFSSL * ssl,
    const unsigned char * in,
    long sz
)
```

This function loads a certificate chain buffer into the WOLFSSL object. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. The buffer must be in PEM format and start with the subject's certificate, ending with the root certificate. Please see the examples for proper usage.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **in** buffer containing certificate to load.
- **sz** size of the certificate located in buffer.

See:

- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_certificate_chain_buffer
- wolfSSL_use_certificate_buffer
- wolfSSL_use_PrivateKey_buffer

Return:

- SSL_SUCCES upon success.
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- BUFFER_E will be returned if a chain buffer is bigger than the receiving buffer.

Example

```
int ret;
byte certChainBuff[...];
WOLFSSL* ssl = 0;
long buffSz = sizeof(certBuff);
...
ret = wolfSSL_use_certificate_chain_buffer(ssl, certChainBuff, buffSz);
if (ret != SSL_SUCCESS) {
    // failed to load certificate chain from buffer
}
```

```
int wolfSSL_UnloadCertsKeys(
    WOLFSSL * ssl
)
```

This function unloads any certificates or keys that SSL owns.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_CTX_UnloadCAs

Return:

- SSL_SUCCESS - returned if the function executed successfully.
- BAD_FUNC_ARG - returned if the WOLFSSL object is NULL.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
...
int unloadKeys = wolfSSL_UnloadCertsKeys(ssl);
```

```
if(unloadKeys != SSL_SUCCESS){
    // Failure case.
}
```

```
int wolfSSL_GetIVSize(
    WOLFSSL * ssl
)
```

Returns the iv_size member of the specs structure held in the WOLFSSL struct.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_GetKeySize
- wolfSSL_GetClientWriteIV
- wolfSSL_GetServerWriteIV

Return:

- iv_size returns the value held in ssl->specs.iv_size.
- BAD_FUNC_ARG returned if the WOLFSSL structure is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
int ivSize;
...
ivSize = wolfSSL_GetIVSize(ssl);

if(ivSize > 0){
    // ivSize holds the specs.iv_size value.
}
```

```
void wolfSSL_KeepArrays(
    WOLFSSL * ssl
)
```

Normally, at the end of the SSL handshake, wolfSSL frees temporary arrays. Calling this function before the handshake begins will prevent wolfSSL from freeing temporary arrays. Temporary arrays may be needed for things such as wolfSSL_get_keys() or PSK hints. When the user is done with temporary arrays, either wolfSSL_FreeArrays() may be called to free the resources immediately, or alternatively the resources will be freed when the associated SSL object is freed.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_FreeArrays

Return: none No return.

Example

```
WOLFSSL* ssl;
...
wolfSSL_KeepArrays(ssl);
```

```
void wolfSSL_FreeArrays(
    WOLFSSL * ssl
)
```

Normally, at the end of the SSL handshake, wolfSSL frees temporary arrays. If wolfSSL_KeepArrays() has been called before the handshake, wolfSSL will not free temporary arrays. This function explicitly frees temporary arrays and should be called when the user is done with temporary arrays and does not want to wait for the SSL object to be freed to free these resources.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_KeepArrays

Return: none No return.

Example

```
WOLFSSL* ssl;
...
wolfSSL_FreeArrays(ssl);
```

```
int wolfSSL_DeriveTlsKeys(
    unsigned char * key_data,
    word32 keyLen,
    const unsigned char * ms,
    word32 msLen,
    const unsigned char * sr,
    const unsigned char * cr,
    int tls1_2,
    int hash_type
)
```

An external facing wrapper to derive TLS Keys.

Parameters:

- **key_data** a byte pointer that is allocateded in DeriveTlsKeys and passed through to wc_PRF to hold the final hash.
- **keyLen** a word32 type that is derived in DeriveTlsKeys from the WOLFSSL structure's specs member.
- **ms** a constant pointer type holding the master secret held in the arrays structure within the WOLFSSL structure.
- **msLen** a word32 type that holds the length of the master secret in an enumerated define, SECRET_LEN.
- **sr** a constant byte pointer to the serverRandom member of the arrays structure within the WOLFSSL structure.
- **cr** a constant byte pointer to the clientRandom member of the arrays structure within the WOLFSSL structure.
- **tls1_2** an integer type returned from IsAtLeastTLSv1_2().
- **hash_type** an integer type held in the WOLFSSL structure.

See:

- wc_PRF
- DeriveTlsKeys
- IsAtLeastTLSv1_2

Return:

- 0 returned on success.
- BUFFER_E returned if the sum of labLen and seedLen (computes total size) exceeds the maximum size.
- MEMORY_E returned if the allocation of memory failed.

Example

```
int DeriveTlsKeys(WOLFSSL* ssl){
int ret;
…
ret = wolfSSL_DeriveTlsKeys(key_data, length, ssl->arrays->masterSecret,
SECRET_LEN, ssl->arrays->clientRandom,
IsAtLeastTLSv1_2(ssl), ssl->specs.mac_algorithm);
…
}
```

```
int wolfSSL_X509_get_ext_by_NID(
    const WOLFSSL_X509 * x,
    int nid,
    int lastpos
)
```

This function looks for and returns the extension index matching the passed in NID value.

Parameters:

- **x** certificate to get parse through for extension.

- **nid** extension OID to be found.
- **lastpos** start search from extension after lastpos. Set to -1 initially.

Return:

- = 0 If successful the extension index is returned.
- -1 If extension is not found or error is encountered.

Example

```
const WOLFSSL_X509* x509;
int lastPos = -1;
int idx;

idx = wolfSSL_X509_get_ext_by_NID(x509, NID_basic_constraints, lastPos);
```

```
void * wolfSSL_X509_get_ext_d2i(
    const WOLFSSL_X509 * x509,
    int nid,
    int * c,
    int * idx
)
```

This function looks for and returns the extension matching the passed in NID value.

Parameters:

- **x509** certificate to get parse through for extension.
- **nid** extension OID to be found.
- **c** if not NULL is set to -2 for multiple extensions found -1 if not found, 0 if found and not critical and 1 if found and critical.
- **idx** if NULL return first extension matched otherwise if not stored in x509 start at idx.

See: wolfSSL_sk_ASN1_OBJECT_free

Return:

- pointer If successful a STACK_OF(WOLFSSL_ASN1_OBJECT) pointer is returned.
- NULL If extension is not found or error is encountered.

Example

```
const WOLFSSL_X509* x509;
int c;
int idx = 0;
STACK_OF(WOLFSSL_ASN1_OBJECT)* sk;
```

```
sk = wolfSSL_X509_get_ext_d2i(x509, NID_basic_constraints, &c, &idx);
//check sk for NULL and then use it. sk needs freed after done.
```

```
int wolfSSL_X509_digest(
    const WOLFSSL_X509 * x509,
    const WOLFSSL_EVP_MD * digest,
    unsigned char * buf,
    unsigned int * len
)
```

This function returns the hash of the DER certificate.

Parameters:

- **x509** certificate to get the hash of.
- **digest** the hash algorithm to use.
- **buf** buffer to hold hash.
- **len** length of buffer.

See: none

Return:

- SSL_SUCCESS On successfully creating a hash.
- SSL_FAILURE Returned on bad input or unsuccessful hash.

Example

```
WOLFSSL_X509* x509;
unsigned char buffer[64];
unsigned int bufferSz;
int ret;

ret = wolfSSL_X509_digest(x509, wolfSSL_EVP_sha256(), buffer, &bufferSz);
//check ret value
```

```
int wolfSSL_use_PrivateKey(
    WOLFSSL * ssl,
    WOLFSSL_EVP_PKEY * pkey
)
```

This is used to set the private key for the WOLFSSL structure.

Parameters:

- **ssl** WOLFSSL structure to set argument in.
- **pkey** private key to use.

See:

- wolfSSL_new
- wolfSSL_free

Return:

- SSL_SUCCESS On successful setting argument.
- SSL_FAILURE If a NULL ssl passed in. All error cases will be negative values.

Example

```
WOLFSSL* ssl;
WOLFSSL_EVP_PKEY* pkey;
int ret;
// create ssl object and set up private key
ret  = wolfSSL_use_PrivateKey(ssl, pkey);
// check ret value
```

```
int wolfSSL_use_PrivateKey_ASN1(
    int pri,
    WOLFSSL * ssl,
    const unsigned char * der,
    long derSz
)
```

This is used to set the private key for the WOLFSSL structure. A DER formatted key buffer is expected.

Parameters:

- **pri** type of private key.
- **ssl** WOLFSSL structure to set argument in.
- **der** buffer holding DER key.
- **derSz** size of der buffer.

See:

- wolfSSL_new
- wolfSSL_free
- wolfSSL_use_PrivateKey

Return:

- SSL_SUCCESS On successful setting parsing and setting the private key.
- SSL_FAILURE If an NULL ssl passed in. All error cases will be negative values.

Example

```
WOLFSSL* ssl;
unsigned char* pkey;
long pkeySz;
int ret;
// create ssl object and set up private key
ret  = wolfSSL_use_PrivateKey_ASN1(1, ssl, pkey, pkeySz);
// check ret value
```

```
int wolfSSL_use_RSAPrivateKey_ASN1(
    WOLFSSL * ssl,
    unsigned char * der,
    long derSz
)
```

This is used to set the private key for the WOLFSSL structure. A DER formatted RSA key buffer is expected.

Parameters:

- **ssl** WOLFSSL structure to set argument in.
- **der** buffer holding DER key.
- **derSz** size of der buffer.

See:

- wolfSSL_new
- wolfSSL_free
- wolfSSL_use_PrivateKey

Return:

- SSL_SUCCESS On successful setting parsing and setting the private key.
- SSL_FAILURE If an NULL ssl passed in. All error cases will be negative values.

Example

```
WOLFSSL* ssl;
unsigned char* pkey;
long pkeySz;
int ret;
// create ssl object and set up RSA private key
ret  = wolfSSL_use_RSAPrivateKey_ASN1(ssl, pkey, pkeySz);
// check ret value
```

```
WOLFSSL_DH * wolfSSL_DSA_dup_DH(
    const WOLFSSL_DSA * r
)
```

This function duplicates the parameters in dsa to a newly created WOLFSSL_DH structure.

Parameters:

- **dsa** WOLFSSL_DSA structure to duplicate.

See: none

Return:

- WOLFSSL_DH If duplicated returns WOLFSSL_DH structure
- NULL upon failure

Example

```
WOLFSSL_DH* dh;
WOLFSSL_DSA* dsa;
// set up dsa
dh = wolfSSL_DSA_dup_DH(dsa);

// check dh is not null
```

```
WOLFSSL_X509 * wolfSSL_d2i_X509_bio(
    WOLFSSL_BIO * bio,
    WOLFSSL_X509 ** x509
)
```

This function get the DER buffer from bio and converts it to a WOLFSSL_X509 structure.

Parameters:

- **bio** pointer to the WOLFSSL_BIO structure that has the DER certificate buffer.
- **x509** pointer that get set to new WOLFSSL_X509 structure created.

See: none

Return:

- pointer returns a WOLFSSL_X509 structure pointer on success.
- Null returns NULL on failure

Example

```
WOLFSSL_BIO* bio;
WOLFSSL_X509* x509;
// load DER into bio
x509 = wolfSSL_d2i_X509_bio(bio, NULL);
Or
wolfSSL_d2i_X509_bio(bio, &x509);
// use x509 returned (check for NULL)
```

```
WOLFSSL_X509 * wolfSSL_PEM_read_bio_X509_AUX(
    WOLFSSL_BIO * bp,
    WOLFSSL_X509 ** x,
    wc_pem_password_cb * cb,
    void * u
)
```

This function behaves the same as wolfSSL_PEM_read_bio_X509. AUX signifies containing extra information such as trusted/rejected use cases and friendly name for human readability.

Parameters:

- **bp** WOLFSSL_BIO structure to get PEM buffer from.
- **x** if setting WOLFSSL_X509 by function side effect.
- **cb** password callback.
- **u** NULL terminated user password.

See: wolfSSL_PEM_read_bio_X509

Return:

- WOLFSSL_X509 on successfully parsing the PEM buffer a WOLFSSL_X509 structure is returned.
- Null if failed to parse PEM buffer.

Example

```
WOLFSSL_BIO* bio;
WOLFSSL_X509* x509;
// setup bio
X509 = wolfSSL_PEM_read_bio_X509_AUX(bio, NULL, NULL, NULL);
//check x509 is not null and then use it
```

```
long wolfSSL_CTX_set_tmp_dh(
    WOLFSSL_CTX * ctx,
    WOLFSSL_DH * dh
)
```

Initializes the WOLFSSL_CTX structure's dh member with the Diffie-Hellman parameters.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **dh** a pointer to a WOLFSSL_DH structure.

See: wolfSSL_BN_bn2bin

Return:

- SSL_SUCCESS returned if the function executed successfully.
- BAD_FUNC_ARG returned if the ctx or dh structures are NULL.
- SSL_FATAL_ERROR returned if there was an error setting a structure value.
- MEMORY_E returned if their was a failure to allocate memory.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL_DH* dh;
…
return wolfSSL_CTX_set_tmp_dh(ctx, dh);
```

```
WOLFSSL_DSA * wolfSSL_PEM_read_bio_DSAparams(
    WOLFSSL_BIO * bp,
    WOLFSSL_DSA ** x,
    wc_pem_password_cb * cb,
    void * u
)
```

This function get the DSA parameters from a PEM buffer in bio.

Parameters:

- **bio** pointer to the WOLFSSL_BIO structure for getting PEM memory pointer.
- **x** pointer to be set to new WOLFSSL_DSA structure.
- **cb** password callback function.
- **u** null terminated password string.

See: none

Return:

- WOLFSSL_DSA on successfully parsing the PEM buffer a WOLFSSL_DSA structure is created and returned.
- Null if failed to parse PEM buffer.

Example

```
WOLFSSL_BIO* bio;
WOLFSSL_DSA* dsa;
// setup bio
dsa = wolfSSL_PEM_read_bio_DSAparams(bio, NULL, NULL, NULL);

// check dsa is not NULL and then use dsa
```

```
WOLF_STACK_OF(
    WOLFSSL_X509
) const
```

This function gets the peer's certificate chain.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_subject_name
- wolfSSL_X509_get_isCA

Return:

- pointer returns a pointer to the peer's Certificate stack.
- NULL returned if no peer certificate.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
wolfSSL_connect(ssl);
STACK_OF(WOLFSSL_X509)* chain = wolfSSL_get_peer_cert_chain(ssl);
ifchain){
    // You have a pointer to the peer certificate chain
}
```

```
char * wolfSSL_X509_get_next_altname(
    WOLFSSL_X509 *
)
```

This function returns the next, if any, altname from the peer certificate.

Parameters:

- **cert** a pointer to the wolfSSL_X509 structure.

See:

- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_subject_name
- wolfSSL_X509_get_ext_d2i

Return:

- NULL if there is not a next altname.
- cert->altNamesNext->name from the WOLFSSL_X509 structure that is a string value from the altName list is returned if it exists.

Warning: The returned value is a bare C string with no length. A dNSName, rfc822Name, or uniformResourceIdentifier SAN may legally contain an embedded NUL byte (RFC 6125 Sec. 6.3 treats such a name as an invalid presented identifier, not a malformed certificate, so the certificate still parses), which silently truncates the returned string. Do not use strlen or strcmp on the result for security comparisons; use wolfSSL_X509_get_ext_d2i and the GENERAL_NAME ASN1_STRING length instead.

Example

```
WOLFSSL_X509 x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                                                    DYNAMIC_TYPE_X509);
…
int x509NextAltName = wolfSSL_X509_get_next_altname(x509);
if(x509NextAltName == NULL){
        //There isn't another alt name
}
```

```
WOLFSSL_ASN1_TIME * wolfSSL_X509_get_notBefore(
    WOLFSSL_X509 *
)
```

The function checks to see if x509 is NULL and if it's not, it returns the notBefore member of the x509 struct.

Parameters:

- **x509** a pointer to the WOLFSSL_X509 struct.

See: wolfSSL_X509_get_notAfter

Return:

- pointer to struct with ASN1_TIME to the notBefore member of the x509 struct.
- NULL the function returns NULL if the x509 structure is NULL.

Example

```
WOLFSSL_X509* x509 = (WOLFSSL_X509)XMALLOC(sizeof(WOLFSSL_X509), NULL,
DYNAMIC_TYPE_X509) ;
…
const WOLFSSL_ASN1_TIME* notAfter = wolfSSL_X509_get_notBefore(x509);
if(notAfter == NULL){
        //The x509 object was NULL
}
```

A.5 wolfSSL Connection, Session, and I/O

A.4.2.95 function wolfSSL_X509_get_notBefore

A.5.1 Functions

	Name
long	**wolfSSL_get_verify_depth**(WOLFSSL * ssl)This function returns the maximum chain depth allowed, which is 9 by default, for a valid session i.e. there is a non_null session object (ssl).
char *	**wolfSSL_get_cipher_list**(int priority)Get the name of cipher at priority level passed in.
int	**wolfSSL_get_ciphers**(char * buf, int len)This function gets the ciphers enabled in wolfSSL.
const char *	**wolfSSL_get_cipher_name**(WOLFSSL * ssl)This function gets the cipher name in the format DHE-RSA by passing through argument to wolfSSL_get_cipher_name_internal.
int	**wolfSSL_get_fd**(const WOLFSSL * ssl)This function returns the read file descriptor (fd) used as the input facility for the SSL connection. Typically this will be a socket file descriptor.
int	**wolfSSL_get_wfd**(const WOLFSSL * ssl)This function returns the write file descriptor (fd) used as the output facility for the SSL connection. Typically this will be a socket file descriptor.
int	**wolfSSL_get_using_nonblock**(WOLFSSL *)This function allows the application to determine if wolfSSL is using non_blocking I/O. If wolfSSL is using non_blocking I/O, this function will return 1, otherwise 0. After an application creates a WOLFSSL object, if it will be used with a non_blocking socket, call wolfSSL_set_using_nonblock() on it. This lets the WOLFSSL object know that receiving EWOULDBLOCK means that the recvfrom call would block rather than that it timed out.

	Name
int	**wolfSSL_write**(WOLFSSL * ssl, const void * data, int sz)This function writes sz bytes from the buffer, data, to the SSL connection, ssl. If necessary, wolfSSL_write() will negotiate an SSL/TLS session if the handshake has not already been performed yet by wolfSSL_connect() or wolfSSL_accept(). When using (D)TLSv1.3 and early data feature is compiled in, this function progresses the handshake only up to the point when it is possible to send data. Next invocations of wolfSSL_Connect()/wolfSSL_Accept()/wolfSSL_read() will complete the handshake. wolfSSL_write() works with both blocking and non_blocking I/O. When the underlying I/O is non_blocking, wolfSSL_write() will return when the underlying I/O could not satisfy the needs of wolfSSL_write() to continue. In this case, a call to wolfSSL_get_error() will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_write() when the underlying I/O is ready. If the underlying I/O is blocking, wolfSSL_write() will only return once the buffer data of size sz has been completely written or an error occurred.
int	**wolfSSL_write_ex**(WOLFSSL * ssl, const void * data, size_t sz, size_t * wr)This function writes sz bytes from the buffer, data, to the SSL connection, ssl, and reports the number of bytes written. It is equivalent to wolfSSL_write() except that the length written is returned through wr and the return value only indicates success or failure. Whether a partial write counts as success depends on WOLFSSL_MODE_ENABLE_PARTIAL_WRITE having been set with wolfSSL_CTX_set_mode(); without it, anything short of the full length is a failure.

	Name
int	**wolfSSL_read**(WOLFSSL * ssl, void * data, int sz)This function reads sz bytes from the SSL session (ssl) internal read buffer into the buffer data. The bytes read are removed from the internal receive buffer. If necessary wolfSSL_read() will negotiate an SSL/TLS session if the handshake has not already been performed yet by wolfSSL_connect() or wolfSSL_accept(). The SSL/TLS protocol uses SSL records which have a maximum size of 16kB (the max record size can be controlled by the MAX_RECORD_SIZE define in /wolfssl/internal.h). As such, wolfSSL needs to read an entire SSL record internally before it is able to process and decrypt the record. Because of this, a call to wolfSSL_read() will only be able to return the maximum buffer size which has been decrypted at the time of calling. There may be additional not_yet_decrypted data waiting in the internal wolfSSL receive buffer which will be retrieved and decrypted with the next call to wolfSSL_read(). If sz is larger than the number of bytes in the internal read buffer, SSL_read() will return the bytes available in the internal read buffer. If no bytes are buffered in the internal read buffer yet, a call to wolfSSL_read() will trigger processing of the next record.
int	**wolfSSL_read_ex**(WOLFSSL * ssl, void * data, size_t sz, size_t * rd)This function reads up to sz bytes of decrypted application data from the SSL connection, ssl, into the buffer, data, and reports the number of bytes read. It is equivalent to wolfSSL_read() except that the length read is returned through rd and the return value only indicates whether any application data was read.

	Name
int	**wolfSSL_peek**(WOLFSSL * ssl, void * data, int sz)This function copies sz bytes from the SSL session (ssl) internal read buffer into the buffer data. This function is identical to wolfSSL_read() except that the data in the internal SSL session receive buffer is not removed or modified. If necessary, like wolfSSL_read(), wolfSSL_peek() will negotiate an SSL/TLS session if the handshake has not already been performed yet by wolfSSL_connect() or wolfSSL_accept(). The SSL/TLS protocol uses SSL records which have a maximum size of 16kB (the max record size can be controlled by the MAX_RECORD_SIZE define in /wolfssl/internal.h). As such, wolfSSL needs to read an entire SSL record internally before it is able to process and decrypt the record. Because of this, a call to wolfSSL_peek() will only be able to return the maximum buffer size which has been decrypted at the time of calling. There may be additional not_yet_decrypted data waiting in the internal wolfSSL receive buffer which will be retrieved and decrypted with the next call to wolfSSL_peek() / wolfSSL_read(). If sz is larger than the number of bytes in the internal read buffer, SSL_peek() will return the bytes available in the internal read buffer. If no bytes are buffered in the internal read buffer yet, a call to wolfSSL_peek() will trigger processing of the next record.
int	**wolfSSL_accept**(WOLFSSL * ssl)This function is called on the server side and waits for an SSL client to initiate the SSL/TLS handshake. When this function is called, the underlying communication channel has already been set up. wolfSSL_accept() works with both blocking and non_blocking I/O. When the underlying I/O is non_blocking, wolfSSL_accept() will return when the underlying I/O could not satisfy the needs of wolfSSL_accept to continue the handshake. In this case, a call to wolfSSL_get_error() will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_accept when data is available to read and wolfSSL will pick up where it left off. When using a non_blocking socket, nothing needs to be done, but select() can be used to check for the required condition. If the underlying I/O is blocking, wolfSSL_accept() will only return once the handshake has been finished or an error occurred.

	Name
int	**wolfDTLS_accept_stateless**(WOLFSSL * ssl)This function statelessly listens for an SSL client to initiate the DTLS handshake. Cookies must be enabled. A cookie_disabled DTLS object is rejected before I/O or callback changes: WOLFSSL_FATAL_ERROR is returned and wolfSSL_get_error() reports BAD_STATE_E. Use wolfSSL_accept() for cookie-disabled connections instead.
int	**wolfSSL_send**(WOLFSSL * ssl, const void * data, int sz, int flags)This function writes sz bytes from the buffer, data, to the SSL connection, ssl, using the specified flags for the underlying write operation. If necessary wolfSSL_send() will negotiate an SSL/TLS session if the handshake has not already been performed yet by wolfSSL_connect() or wolfSSL_accept(). wolfSSL_send() works with both blocking and non_blocking I/O. When the underlying I/O is non_blocking, wolfSSL_send() will return when the underlying I/O could not satisfy the needs of wolfSSL_send to continue. In this case, a call to wolfSSL_get_error() will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_send() when the underlying I/O is ready. If the underlying I/O is blocking, wolfSSL_send() will only return once the buffer data of size sz has been completely written or an error occurred.

	Name
int	**wolfSSL_recv**(WOLFSSL * ssl, void * data, int sz, int flags)This function reads sz bytes from the SSL session (ssl) internal read buffer into the buffer data using the specified flags for the underlying recv operation. The bytes read are removed from the internal receive buffer. This function is identical to wolfSSL_read() except that it allows the application to set the recv flags for the underlying read operation. If necessary wolfSSL_recv() will negotiate an SSL/TLS session if the handshake has not already been performed yet by wolfSSL_connect() or wolfSSL_accept(). The SSL/TLS protocol uses SSL records which have a maximum size of 16kB (the max record size can be controlled by the MAX_RECORD_SIZE define in /wolfssl/internal.h). As such, wolfSSL needs to read an entire SSL record internally before it is able to process and decrypt the record. Because of this, a call to wolfSSL_recv() will only be able to return the maximum buffer size which has been decrypted at the time of calling. There may be additional not_yet_decrypted data waiting in the internal wolfSSL receive buffer which will be retrieved and decrypted with the next call to wolfSSL_recv(). If sz is larger than the number of bytes in the internal read buffer, SSL_recv() will return the bytes available in the internal read buffer. If no bytes are buffered in the internal read buffer yet, a call to wolfSSL_recv() will trigger processing of the next record.
int	**wolfSSL_get_alert_history**(WOLFSSL * ssl, WOLFSSL_ALERT_HISTORY * h)This function gets the alert history.
WOLFSSL_SESSION *	**wolfSSL_get_session**(WOLFSSL * ssl)Returns a pointer to the current session (WOLFSSL_SESSION) used in ssl. This is a non_persistent pointer to the WOLFSSL_SESSION object, freed when wolfSSL_free is called, so the call should only be used to inspect or modify the current session. For session resumption use wolfSSL_get1_session(), which returns a reference the caller owns and releases with wolfSSL_SESSION_free().

	Name
void	**wolfSSL_flush_sessions**(WOLFSSL_CTX * ctx, long tm)This function flushes session from the session cache which have expired. The time, tm, is used for the time comparison. Note that wolfSSL currently uses a static table for sessions, so no flushing is needed. As such, this function is currently just a stub. This function provides OpenSSL compatibility (SSL_flush_sessions) when wolfSSL is compiled with the OpenSSL compatibility layer.
int	**wolfSSL_GetSessionIndex**(WOLFSSL * ssl)This function gets the session index of the WOLFSSL structure.
int	**wolfSSL_GetSessionAtIndex**(int index, WOLFSSL_SESSION * session)This function gets the session at specified index of the session cache and copies it into memory. The WOLFSSL_SESSION structure holds the session information. The copy is independent of the cache entry: it does not share the ticket buffer, peer certificate or ex_data with the cache, and it stays valid after the cache entry is overwritten or evicted. The caller owns the copy and releases it with wolfSSL_SESSION_free().
WOLFSSL_X509_CHAIN *	**wolfSSL_SESSION_get_peer_chain**(WOLFSSL_SESSION * session)Returns the peer certificate chain from the WOLFSSL_SESSION struct.
int	**wolfSSL_pending**(WOLFSSL * ssl)This function returns the number of bytes which are buffered and available in the SSL object to be read by wolfSSL_read().
int	**wolfSSL_save_session_cache**(const char * fname)This function persists the session cache to file. It doesn't use memsave because of additional memory use.
int	**wolfSSL_restore_session_cache**(const char * fname)This function restores the persistent session cache from file. It does not use memstore because of additional memory use.
int	**wolfSSL_memsave_session_cache**(void * mem, int sz)This function persists session cache to memory.
int	**wolfSSL_memrestore_session_cache**(const void * mem, int sz)This function restores the persistent session cache from memory.
int	**wolfSSL_get_session_cache_memsize**(void)This function returns how large the session cache save buffer should be.

	Name
int	**wolfSSL_session_reused**(WOLFSSL * ssl)This function returns the resuming member of the options struct. The flag indicates whether or not to reuse a session. If not, a new session must be established.
const char *	**wolfSSL_get_version**(WOLFSSL * ssl)Returns the SSL version being used as a string.
int	**wolfSSL_get_current_cipher_suite**(WOLFSSL * ssl)Returns the current cipher suit an ssl session is using.
WOLFSSL_CIPHER *	**wolfSSL_get_current_cipher**(WOLFSSL * ssl)This function returns a pointer to the current cipher in the ssl session. The cipher is filled in from the session on each call, so call it again after the session changes, e.g. after the handshake.
const char *	**wolfSSL_CIPHER_get_name**(const WOLFSSL_CIPHER * cipher)This function matches the cipher suite in the SSL object with the available suites and returns the string representation.
const char *	**wolfSSL_get_cipher**(WOLFSSL *)This function matches the cipher suite in the SSL object with the available suites.
int	**wolfSSL_BIO_get_mem_data**(WOLFSSL_BIO * bio, void * p)This is used to set a byte pointer to the start of the internal memory buffer.
void	**wolfSSL_BIO_set_init**(WOLFSSL_BIO * bio, int init)This is used to set the init flag of a BIO, indicating whether the BIO has been initialised and is ready for use. Typically called from a custom BIO create callback.
int	**wolfSSL_BIO_get_init**(WOLFSSL_BIO * bio)This is used to retrieve the init flag of a BIO, indicating whether the BIO has been initialised and is ready for use.
long	**wolfSSL_BIO_set_fd**(WOLFSSL_BIO * b, int fd, int flag)Sets the file descriptor for bio to use.
int	**wolfSSL_BIO_set_close**(WOLFSSL_BIO * b, long flag)Sets the close flag, used to indicate that the i/o stream should be closed when the BIO is freed.
WOLFSSL_BIO_METHOD *	**wolfSSL_BIO_s_socket**(void)This is used to get a BIO_SOCKET type WOLFSSL_BIO_METHOD.
int	**wolfSSL_BIO_set_write_buf_size**(WOLFSSL_BIO * b, long size)This is used to set the size of write buffer for a WOLFSSL_BIO. If write buffer has been previously set this function will free it when resetting the size. It is similar to wolfSSL_BIO_reset in that it resets read and write indexes to 0.

	Name
int	**wolfSSL_BIO_make_bio_pair**(WOLFSSL_BIO * b1, WOLFSSL_BIO * b2)This is used to pair two bios together. A pair of bios acts similar to a two way pipe writing to one can be read by the other and vice versa. It is expected that both bios be in the same thread, this function is not thread safe. Freeing one of the two bios removes both from being paired. If a write buffer size was not previously set for either of the bios it is set to a default size of 17000 (WOLFSSL_BIO_SIZE) before being paired.
int	**wolfSSL_BIO_ctrl_reset_read_request**(WOLFSSL_BIO * b)This is used to set the read request flag back to 0.
int	**wolfSSL_BIO_nread0**(WOLFSSL_BIO * bio, char ** buf)This is used to get a buffer pointer for reading from. Unlike wolfSSL_BIO_nread the internal read index is not advanced by the number returned from the function call. Reading past the value returned can result in reading out of array bounds.
int	**wolfSSL_BIO_nread**(WOLFSSL_BIO * bio, char ** buf, int num)This is used to get a buffer pointer for reading from. The internal read index is advanced by the number returned from the function call with buf being pointed to the beginning of the buffer to read from. In the case that less bytes are in the read buffer than the value requested with num the lesser value is returned. Reading past the value returned can result in reading out of array bounds.
int	**wolfSSL_BIO_nwrite**(WOLFSSL_BIO * bio, char ** buf, int num)Gets a pointer to the buffer for writing as many bytes as returned by the function. Writing more bytes to the pointer returned then the value returned can result in writing out of bounds.
int	**wolfSSL_BIO_reset**(WOLFSSL_BIO * bio)Resets bio to an initial state. As an example for type BIO_BIO this resets the read and write index.
int	**wolfSSL_BIO_seek**(WOLFSSL_BIO * bio, int ofs)This function adjusts the file pointer to the offset given. This is the offset from the head of the file.
int	**wolfSSL_BIO_write_filename**(WOLFSSL_BIO * bio, char * name)This is used to set and write to a file. WIll overwrite any data currently in the file and is set to close the file when the bio is freed.

	Name
long	**wolfSSL_BIO_set_mem_eof_return**(WOLFSSL_BIO * bio, int v)This is used to set the end of file value. Common value is -1 so as not to get confused with expected positive values.
long	**wolfSSL_BIO_get_mem_ptr**(WOLFSSL_BIO * bio, WOLFSSL_BUF_MEM ** m)This is a getter function for WOLFSSL_BIO memory pointer.
const char *	**wolfSSL_lib_version**(void)This function returns the current library version.
word32	**wolfSSL_lib_version_hex**(void)This function returns the current library version in hexadecimal notation.
int	**wolfSSL_negotiate**(WOLFSSL * ssl)Performs the actual connect or accept based on the side of the SSL method. If called from the client side then an wolfSSL_connect() is done while a wolfSSL_accept() is performed if called from the server side.
int	**wolfSSL_connect_cert**(WOLFSSL * ssl)This function is called on the client side and initiates an SSL/TLS handshake with a server only long enough to get the peer's certificate chain. When this function is called, the underlying communication channel has already been set up. wolfSSL_connect_cert() works with both blocking and non_blocking I/O. When the underlying I/O is non_blocking, wolfSSL_connect_cert() will return when the underlying I/O could not satisfy the needs of wolfSSL_connect_cert() to continue the handshake. In this case, a call to wolfSSL_get_error() will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_connect_cert() when the underlying I/O is ready and wolfSSL will pick up where it left off. When using a non_blocking socket, nothing needs to be done, but select() can be used to check for the required condition. If the underlying I/O is blocking, wolfSSL_connect_cert() will only return once the peer's certificate chain has been received.
int	**wolfSSL_writev**(WOLFSSL * ssl, const struct iovec * iov, int iovcnt)Simulates writev semantics but doesn't actually do block at a time because of SSL_write() behavior and because front adds may be small. Makes porting into software that uses writev easier.
unsigned char	**wolfSSL_SNI_Status**(WOLFSSL * ssl, unsigned char type)This function gets the status of an SNI object.

	Name
int	**wolfSSL_UseSecureRenegotiation**(WOLFSSL * ssl)This function forces secure renegotiation for the supplied WOLFSSL structure. This is not recommended.
int	**wolfSSL_Rehandshake**(WOLFSSL * ssl)This function executes a secure renegotiation handshake; this is user forced as wolfSSL discourages this functionality.
int	**wolfSSL_UseSessionTicket**(WOLFSSL * ssl)Force provided WOLFSSL structure to use session ticket. The constant HAVE_SESSION_TICKET should be defined and the constant NO_WOLFSSL_CLIENT should not be defined to use this function.
int	**wolfSSL_get_SessionTicket**(WOLFSSL * ssl, unsigned char * buf, word32 * bufSz)This function copies the ticket member of the Session structure to the buffer. If buf is NULL and bufSz is non-NULL, bufSz will be set to the ticket length.
int	**wolfSSL_set_SessionTicket**(WOLFSSL * ssl, const unsigned char * buf, word32 bufSz)This function sets the ticket member of the WOLFSSL_SESSION structure within the WOLFSSL struct. The buffer passed into the function is copied to memory.
int	**wolfSSL_PrintSessionStats**(void)This function prints the statistics from the session.
int	**wolfSSL_get_session_stats**(unsigned int * active, unsigned int * total, unsigned int * peak, unsigned int * maxSessions)This function gets the statistics for the session.
long	**wolfSSL_BIO_set_fp**(WOLFSSL_BIO * bio, XFILE fp, int c)This is used to set the internal file pointer for a BIO.
long	**wolfSSL_BIO_get_fp**(WOLFSSL_BIO * bio, XFILE * fp)This is used to get the internal file pointer for a BIO.
size_t	**wolfSSL_BIO_ctrl_pending**(WOLFSSL_BIO * b)Gets the number of pending bytes to read. If BIO type is BIO_BIO then is the number to read from pair. If BIO contains an SSL object then is pending data from SSL object (wolfSSL_pending(ssl)). If is BIO_MEMORY type then returns the size of memory buffer.
int	**wolfSSL_set_jobject**(WOLFSSL * ssl, void * objPtr)This function sets the jObjectRef member of the WOLFSSL structure.
void *	**wolfSSL_get_jobject**(WOLFSSL * ssl)This function returns the jObjectRef member of the WOLFSSL structure.

	Name
int	**wolfSSL_connect**(WOLFSSL * ssl)This function is called on the client side and initiates an SSL/TLS handshake with a server. When this function is called, the underlying communication channel has already been set up. wolfSSL_connect() works with both blocking and non_blocking I/O. When the underlying I/O is non_blocking, wolfSSL_connect() will return when the underlying I/O could not satisfy the needs of wolfSSL_connect to continue the handshake. In this case, a call to wolfSSL_get_error() will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_connect() when the underlying I/O is ready and wolfSSL will pick up where it left off. When using a non_blocking socket, nothing needs to be done, but select() can be used to check for the required condition. If the underlying I/O is blocking, wolfSSL_connect() will only return once the handshake has been finished or an error occurred. wolfSSL takes a different approach to certificate verification than OpenSSL does. The default policy for the client is to verify the server, this means that if you don't load CAs to verify the server you'll get a connect error, unable to verify (_155). It you want to mimic OpenSSL behavior of having SSL_connect succeed even if verifying the server fails and reducing security you can do this by calling: SSL_CTX_set_verify(ctx, SSL_VERIFY_NONE, 0); before calling SSL_new(); Though it's not recommended.
int	**wolfSSL_update_keys**(WOLFSSL * ssl)This function is called on a TLS v1.3 client or server wolfSSL to force the rollover of keys. A KeyUpdate message is sent to the peer and new keys are calculated for encryption. The peer will send back a KeyUpdate message and the new decryption keys will then be calculated. This function can only be called after a handshake has been completed.
int	**wolfSSL_key_update_response**(WOLFSSL * ssl, int * required)This function is called on a TLS v1.3 client or server wolfSSL to determine whether a rollover of keys is in progress. When wolfSSL_update_keys() is called, a KeyUpdate message is sent and the encryption key is updated. The decryption key is updated when the response is received.

	Name
int	**wolfSSL_request_certificate**(WOLFSSL * ssl)This function requests a client certificate from the TLS v1.3 client. This is useful when a web server is serving some pages that require client authentication and others that don't. A maximum of 256 requests can be sent on a connection.
int	**wolfSSL_connect_TLSv13**(WOLFSSL * ssl)This function is called on the client side and initiates a TLS v1.3 handshake with a server. When this function is called, the underlying communication channel has already been set up. wolfSSL_connect() works with both blocking and non_blocking I/O. When the underlying I/O is non_blocking, wolfSSL_connect() will return when the underlying I/O could not satisfy the needs of wolfSSL_connect to continue the handshake. In this case, a call to wolfSSL_get_error() will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_connect() when the underlying I/O is ready and wolfSSL will pick up where it left off. When using a non_blocking socket, nothing needs to be done, but select() can be used to check for the required condition. If the underlying I/O is blocking, wolfSSL_connect() will only return once the handshake has been finished or an error occurred. wolfSSL takes a different approach to certificate verification than OpenSSL does. The default policy for the client is to verify the server, this means that if you don't load CAs to verify the server you'll get a connect error, unable to verify (_155). It you want to mimic OpenSSL behavior of having SSL_connect succeed even if verifying the server fails and reducing security you can do this by calling: SSL_CTX_set_verify(ctx, SSL_VERIFY_NONE, 0); before calling SSL_new(); Though it's not recommended.

	Name
	wolfSSL_accept_TLSv13(WOLFSSL * ssl)This function is called on the server side and waits for a SSL/TLS client to initiate the SSL/TLS handshake. When this function is called, the underlying communication channel has already been set up. wolfSSL_accept() works with both blocking and non_blocking I/O. When the underlying I/O is non_blocking, wolfSSL_accept() will return when the underlying I/O could not satisfy the needs of wolfSSL_accept to continue the handshake. In this case, a call to wolfSSL_get_error() will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_accept when data is available to read and wolfSSL will pick up where it left off. When using a non_blocking socket, nothing needs to be done, but select() can be used to check for the required condition. If the underlying I/O is blocking, wolfSSL_accept() will only return once the handshake has been finished or an error occurred. Call this function when expecting a TLS v1.3 connection though older version ClientHello messages are supported.
int	**wolfSSL_write_early_data**(WOLFSSL * ssl, const void * data, int sz, int * outSz)This function writes early data to the server on resumption. Call this function before wolfSSL_connect() or wolfSSL_connect_TLSv13(). This function is only used with clients.
int	**wolfSSL_read_early_data**(WOLFSSL * ssl, void * data, int sz, int * outSz)This function reads any early data from a client on resumption. Call this function instead of wolfSSL_accept() or wolfSSL_accept_TLSv13() to accept a client and read any early data in the handshake. The function should be invoked until wolfSSL_is_init_finished() returns true. Early data may be sent by the client in multiple messages. If there is no early data then the handshake will be processed as normal. This function is only used with servers.
int	**wolfSSL_inject**(WOLFSSL * ssl, const void * data, int sz)This function is called to inject data into the WOLFSSL object. This is useful when data needs to be read from a single place and demultiplexed into multiple connections. The caller should then call wolfSSL_read() to extract the plaintext data from the WOLFSSL object.

	Name
void *	**wolfSSL_GetIOReadCtx**(WOLFSSL * ssl)This function returns the IOCB_ReadCtx member of the WOLFSSL struct.
void *	**wolfSSL_GetIOWriteCtx**(WOLFSSL * ssl)This function returns the IOCB_WriteCtx member of the WOLFSSL structure.
void	**wolfSSL_SetIO_NetX**(WOLFSSL * ssl, NX_TCP_SOCKET * nxsocket, ULONG waitoption)This function sets the nxTcpSocket and nxWait members of the nxCtx struct within the WOLFSSL structure.
void	**wolfSSL_SetIO_NetX_Dtls**(WOLFSSL * ssl, NX_UDP_SOCKET * nxsocket, NXD_ADDRESS nxdip, USHORT nxport, ULONG waitoption)This function configures the NetX Duo UDP context for a DTLS session. It stores the UDP socket, destination IP address (by value), destination port, and wait option into the WOLFSSL nxCtx structure. Requires WOLFSSL_NETX_DUO to be defined (ThreadX NetX Duo SDK).
int	**wolfIO_Select**(SOCKET_T sockfd, int to_sec)Waits for socket to be ready for I/O with timeout.
int	**wolfIO_TcpConnect**(SOCKET_T * sockfd, const char * ip, unsigned short port, int to_sec)Connects to TCP server with timeout.
int	**wolfIO_TcpAccept**(SOCKET_T sockfd, SOCKADDR * peer_addr, XSOCKLENT * peer_len)Accepts TCP connection.
int	**wolfIO_TcpBind**(SOCKET_T * sockfd, word16 port)Binds TCP socket to port.
int	**wolfIO_Send**(SOCKET_T sd, char * buf, int sz, int wrFlags)Sends data on socket.
int	**wolfIO_Recv**(SOCKET_T sd, char * buf, int sz, int rdFlags)Receives data from socket.
int	**wolfIO_SendTo**(SOCKET_T sd, WOLFSSL_BIO_ADDR * addr, char * buf, int sz, int wrFlags)Sends datagram to address.
int	**wolfIO_RecvFrom**(SOCKET_T sd, WOLFSSL_BIO_ADDR * addr, char * buf, int sz, int rdFlags)Receives datagram from address.
int	**wolfSSL_BioSend**(WOLFSSL * ssl, char * buf, int sz, void * ctx)BIO send callback.
int	**wolfSSL_BioReceive**(WOLFSSL * ssl, char * buf, int sz, void * ctx)BIO receive callback.
int	**EmbedReceiveFromMcast**(WOLFSSL * ssl, char * buf, int sz, void * ctx)Receives multicast datagram.
int	**wolfIO_HttpBuildRequestOcsp**(const char * domainName, const char * path, int ocspReqSz, unsigned char * buf, int bufSize)Builds HTTP OCSP request.

	Name
int	**wolfIO_HttpProcessResponseOcspGenericIO**(WolfSSLGenericI ioCb, void * ioCbCtx, unsigned char ** respBuf, unsigned char * httpBuf, int httpBufSz, void * heap)Processes HTTP OCSP response with generic I/O.
int	**wolfIO_HttpProcessResponseOcsp**(int sfd, unsigned char ** respBuf, unsigned char * httpBuf, int httpBufSz, void * heap)Processes HTTP OCSP response.
int	**EmbedOcspLookup**(void * ctx, const char * url, int urlSz, byte * ocspReqBuf, int ocspReqSz, byte ** ocspRespBuf)OCSP lookup callback.
int	**wolfIO_HttpBuildRequestCrl**(const char * url, int urlSz, const char * domainName, unsigned char * buf, int bufSize)Builds HTTP CRL request.
int	**wolfIO_HttpProcessResponseCrl**(WOLFSSL_CRL * crl, int sfd, unsigned char * httpBuf, int httpBufSz)Processes HTTP CRL response.
int	**EmbedCrlLookup**(WOLFSSL_CRL * crl, const char * url, int urlSz)CRL lookup callback.
int	**wolfIO_DecodeUrl**(const char * url, int urlSz, char * outName, char * outPath, unsigned short * outPort)Decodes URL into components.
int	**wolfIO_HttpBuildRequest**(const char * reqType, const char * domainName, const char * path, int pathLen, int reqSz, const char * contentType, unsigned char * buf, int bufSize)Builds generic HTTP request.
int	**wolfIO_HttpProcessResponseGenericIO**(WolfSSLGenericIORecv ioCb, void * ioCbCtx, const char ** appStrList, unsigned char ** respBuf, unsigned char * httpBuf, int httpBufSz, int dynType, void * heap)Processes HTTP response with generic I/O.
int	**wolfIO_HttpProcessResponse**(int sfd, const char ** appStrList, unsigned char ** respBuf, unsigned char * httpBuf, int httpBufSz, int dynType, void * heap)Processes HTTP response.
void	**wolfSSL_CTX_SetIOSend**(WOLFSSL_CTX * ctx, CallbackIOSend CBIOSend)Sets I/O send callback for context.
void	**wolfSSL_SSLSetIORecv**(WOLFSSL * ssl, CallbackIORecv CBIORecv)Sets I/O receive callback for SSL object.
void	**wolfSSL_SSLSetIOSend**(WOLFSSL * ssl, CallbackIOSend CBIOSend)Sets I/O send callback for SSL object.
void	**wolfSSL_SetIO_Mynewt**(WOLFSSL * ssl, struct mn_socket * mnSocket, struct mn_sockaddr_in * mnSockAddrIn)Sets I/O for Mynewt platform.

	Name
int	**wolfSSL_SetIO_LwIP**(WOLFSSL * ssl, void * pcb, tcp_recv_fn recv, tcp_sent_fn sent, void * arg)Sets I/O for LwIP platform.
void	**wolfSSL_SetCookieCtx**(WOLFSSL * ssl, void * ctx)Sets cookie context for DTLS.
void	**wolfSSL_CTX_SetIOGetPeer**(WOLFSSL_CTX * ctx, CallbackGetPeer cb)Sets get peer callback for context.
void	**wolfSSL_CTX_SetIOSetPeer**(WOLFSSL_CTX * ctx, CallbackSetPeer cb)Sets set peer callback for context.
int	**EmbedGetPeer**(WOLFSSL * ssl, char * ip, int * ipSz, unsigned short * port, int * fam)Gets peer information.
int	**EmbedSetPeer**(WOLFSSL * ssl, char * ip, int ipSz, unsigned short port, int fam)Sets peer information.

A.5.2 Functions Documentation

```
long wolfSSL_get_verify_depth(
    WOLFSSL * ssl
)
```

This function returns the maximum chain depth allowed, which is 9 by default, for a valid session i.e. there is a non-null session object (ssl).

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_CTX_get_verify_depth

Return:

- MAX_CHAIN_DEPTH returned if the WOLFSSL structure is not NULL. By default the value is 9.
- BAD_FUNC_ARG returned if the WOLFSSL structure is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
long sslDep = wolfSSL_get_verify_depth(ssl);

if(sslDep > EXPECTED){
    // The verified depth is greater than what was expected
} else {
    // The verified depth is smaller or equal to the expected value
}
```

```
char * wolfSSL_get_cipher_list(
    int priority
)
```

Get the name of cipher at priority level passed in.

Parameters:

- **priority** Integer representing the priority level of a cipher.

See:

- wolfSSL_CIPHER_get_name
- wolfSSL_get_current_cipher

Return:

- string Success
- 0 Priority is either out of bounds or not valid.

Example

```
printf("The cipher at 1 is %s", wolfSSL_get_cipher_list(1));
```

```
int wolfSSL_get_ciphers(
    char * buf,
    int len
)
```

This function gets the ciphers enabled in wolfSSL.

Parameters:

- **buf** a char pointer representing the buffer.
- **len** the length of the buffer.

See:

- GetCipherNames
- wolfSSL_get_cipher_list
- ShowCiphers

Return:

- SSL_SUCCESS returned if the function executed without error.
- BAD_FUNC_ARG returned if the buf parameter was NULL or if the len argument was less than or equal to zero.
- BUFFER_E returned if the buffer is not large enough and will overflow.

Example

```
static void ShowCiphers(void){
char* ciphers;
int ret = wolfSSL_get_ciphers(ciphers, (int)sizeof(ciphers));

if(ret == SSL_SUCCES){
        printf("%s\n", ciphers);
    }
}
```

```
const char * wolfSSL_get_cipher_name(
    WOLFSSL * ssl
)
```

This function gets the cipher name in the format DHE-RSA by passing through argument to wolfSSL_get_cipher_name_internal.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_CIPHER_get_name
- wolfSSL_get_current_cipher
- wolfSSL_get_cipher_name_internal

Return:

- string This function returns the string representation of the cipher suite that was matched.
- NULL error or cipher not found.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
…
char* cipherS = wolfSSL_get_cipher_name(ssl);

if(cipher == NULL){
    // There was not a cipher suite matched
} else {
    // There was a cipher suite matched
    printf("%s\n", cipherS);
}
```

```
int wolfSSL_get_fd(
    const WOLFSSL * ssl
)
```

This function returns the read file descriptor (fd) used as the input facility for the SSL connection. Typically this will be a socket file descriptor.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See:

- wolfSSL_set_fd
- wolfSSL_set_read_fd
- wolfSSL_set_write_fd

Return: fd If successful the call will return the SSL session file descriptor.

Example

```
int sockfd;
WOLFSSL* ssl = 0;
...
sockfd = wolfSSL_get_fd(ssl);
...
```

```
int wolfSSL_get_wfd(
    const WOLFSSL * ssl
)
```

This function returns the write file descriptor (fd) used as the output facility for the SSL connection. Typically this will be a socket file descriptor.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See:

- wolfSSL_set_fd
- wolfSSL_set_read_fd
- wolfSSL_set_write_fd

Return: fd If successful the call will return the SSL session file descriptor.

Example

```
int sockfd;
WOLFSSL* ssl = 0;
...
sockfd = wolfSSL_get_wfd(ssl);
...
```

```
int wolfSSL_get_using_nonblock(
    WOLFSSL *
)
```

This function allows the application to determine if wolfSSL is using non-blocking I/O. If wolfSSL is using non-blocking I/O, this function will return 1, otherwise 0. After an application creates a WOLFSSL object, if it will be used with a non-blocking socket, call wolfSSL_set_using_nonblock() on it. This lets the WOLFSSL object know that receiving EWOULDBLOCK means that the recvfrom call would block rather than that it timed out.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See: wolfSSL_set_session

Return:

- 0 underlying I/O is blocking.
- 1 underlying I/O is non-blocking.

Example

```
int ret = 0;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_get_using_nonblock(ssl);
if (ret == 1) {
    // underlying I/O is non-blocking
}
...
```

```
int wolfSSL_write(
    WOLFSSL * ssl,
    const void * data,
    int sz
)
```

This function writes sz bytes from the buffer, data, to the SSL connection, ssl. If necessary, wolfSSL_write() will negotiate an SSL/TLS session if the handshake has not already been performed yet by wolfSSL_connect() or wolfSSL_accept(). When using (D)TLSv1.3 and early data feature is compiled in, this function progresses the handshake only up to the point when it is possible to send data. Next invocations of wolfSSL_Connect()/wolfSSL_Accept()/wolfSSL_read() will complete the handshake. wolfSSL_write() works with both blocking and non-blocking I/O. When the underlying I/O is non-blocking,

wolfSSL_write() will return when the underlying I/O could not satisfy the needs of wolfSSL_write() to continue. In this case, a call to wolfSSL_get_error() will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_write() when the underlying I/O is ready. If the underlying I/O is blocking, wolfSSL_write() will only return once the buffer data of size sz has been completely written or an error occurred.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **data** data buffer which will be sent to peer.
- **sz** size, in bytes, of data to send to the peer (data).

See:

- wolfSSL_send
- wolfSSL_read
- wolfSSL_recv

Return:

- 0 the number of bytes written upon success.
- 0 will be returned upon failure. Call wolfSSL_get_error() for the specific error code.
- SSL_FATAL_ERROR will be returned upon failure when either an error occurred or, when using non-blocking sockets, the SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE error was received and and the application needs to call wolfSSL_write() again. Use wolfSSL_get_error() to get a specific error code.
- BAD_FUNC_ARG will be returned when ssl or data is NULL, or sz is negative.

Example

```
WOLFSSL* ssl = 0;
char msg[64] = “hello wolfssl!”;
int msgSz = (int)strlen(msg);
int flags;
int ret;
...

ret = wolfSSL_write(ssl, msg, msgSz);
if (ret <= 0) {
    // wolfSSL_write() failed, call wolfSSL_get_error()
}
```

```
int wolfSSL_write_ex(
    WOLFSSL * ssl,
    const void * data,
    size_t sz,
    size_t * wr
```

)

This function writes sz bytes from the buffer, data, to the SSL connection, ssl, and reports the number of bytes written. It is equivalent to wolfSSL_write() except that the length written is returned through wr and the return value only indicates success or failure. Whether a partial write counts as success depends on WOLFSSL_MODE_ENABLE_PARTIAL_WRITE having been set with wolfSSL_CTX_set_mode(); without it, anything short of the full length is a failure.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **data** data buffer to write to the SSL connection.
- **sz** number of bytes to write.
- **wr** pointer that receives the number of bytes written. May be NULL. Set to zero before anything else is done, so it reads as zero even when the call fails. This differs from wolfSSL_read_ex(), which leaves its count alone unless data was read.

See:

- wolfSSL_write
- wolfSSL_read_ex
- wolfSSL_get_error

Return:

- 1 on success.
- 0 on failure. Call wolfSSL_get_error() for the reason.
- BAD_FUNC_ARG when ssl is NULL.

Example

```
WOLFSSL* ssl = 0;
char msg[] = "hello wolfssl!";
size_t written = 0;
...

if (wolfSSL_write_ex(ssl, msg, sizeof(msg), &written) != 1) {
    // handle the failure, see wolfSSL_get_error()
}
```

```
int wolfSSL_read(
    WOLFSSL * ssl,
    void * data,
    int sz
)
```

This function reads sz bytes from the SSL session (ssl) internal read buffer into the buffer data. The bytes read are removed from the internal receive buffer. If necessary wolfSSL_read() will negotiate an SSL/TLS session if the handshake has not already been performed yet by wolfSSL_connect() or wolfSSL_accept(). The SSL/TLS protocol uses SSL records which have a maximum size of 16kB (the max

record size can be controlled by the MAX_RECORD_SIZE define in /wolfssl/internal.h). As such, wolfSSL needs to read an entire SSL record internally before it is able to process and decrypt the record. Because of this, a call to wolfSSL_read() will only be able to return the maximum buffer size which has been decrypted at the time of calling. There may be additional not-yet-decrypted data waiting in the internal wolfSSL receive buffer which will be retrieved and decrypted with the next call to wolfSSL_read(). If sz is larger than the number of bytes in the internal read buffer, SSL_read() will return the bytes available in the internal read buffer. If no bytes are buffered in the internal read buffer yet, a call to wolfSSL_read() will trigger processing of the next record.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **data** buffer where wolfSSL_read() will place data read.
- **sz** number of bytes to read into data.

See:

- wolfSSL_recv
- wolfSSL_write
- wolfSSL_peek
- wolfSSL_pending

Return:

- 0 the number of bytes read upon success.
- 0 will be returned upon failure. This may be caused by a either a clean (close notify alert) shutdown or just that the peer closed the connection. Call wolfSSL_get_error() for the specific error code.
- SSL_FATAL_ERROR will be returned upon failure when either an error occurred or, when using non-blocking sockets, the SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE error was received and and the application needs to call wolfSSL_read() again. Use wolfSSL_get_error() to get a specific error code.
- BAD_FUNC_ARG will be returned when ssl or data is NULL, or sz is negative.

Example

```
WOLFSSL* ssl = 0;
char reply[1024];
...

input = wolfSSL_read(ssl, reply, sizeof(reply));
if (input > 0) {
    // “input” number of bytes returned into buffer “reply”
}

See wolfSSL examples (client, server, echoclient, echoserver) for more
complete examples of wolfSSL_read().
```

```
int wolfSSL_read_ex(
    WOLFSSL * ssl,
    void * data,
    size_t sz,
    size_t * rd
)
```

This function reads up to sz bytes of decrypted application data from the SSL connection, ssl, into the buffer, data, and reports the number of bytes read. It is equivalent to wolfSSL_read() except that the length read is returned through rd and the return value only indicates whether any application data was read.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **data** buffer to hold the data read.
- **sz** size of the buffer in bytes.
- **rd** pointer that receives the number of bytes read. May be NULL and is only set when data was read, so it keeps whatever the caller left in it when the call fails. This differs from wolfSSL_write_ex(), which clears its count first.

See:

- wolfSSL_read
- wolfSSL_write_ex
- wolfSSL_pending
- wolfSSL_get_error

Return:

- 1 when application data was read.
- 0 when no application data was read. Call wolfSSL_get_error() for the reason.
- BAD_FUNC_ARG when ssl is NULL.

Example

```
WOLFSSL* ssl = 0;
char reply[1024];
size_t bytesRead = 0;
...

if (wolfSSL_read_ex(ssl, reply, sizeof(reply), &bytesRead) == 1) {
    // "bytesRead" bytes returned into buffer "reply"
}
```

```
int wolfSSL_peek(
    WOLFSSL * ssl,
    void * data,
    int sz
)
```

This function copies sz bytes from the SSL session (ssl) internal read buffer into the buffer data. This function is identical to wolfSSL_read() except that the data in the internal SSL session receive buffer is not removed or modified. If necessary, like wolfSSL_read(), wolfSSL_peek() will negotiate an SSL/TLS session if the handshake has not already been performed yet by wolfSSL_connect() or wolfSSL_accept(). The SSL/TLS protocol uses SSL records which have a maximum size of 16kB (the max record size can be controlled by the MAX_RECORD_SIZE define in /wolfssl/internal.h). As such, wolfSSL needs to read an entire SSL record internally before it is able to process and decrypt the record. Because of this, a call to wolfSSL_peek() will only be able to return the maximum buffer size which has been decrypted at the time of calling. There may be additional not_yet_decrypted data waiting in the internal wolfSSL receive buffer which will be retrieved and decrypted with the next call to wolfSSL_peek() / wolfSSL_read(). If sz is larger than the number of bytes in the internal read buffer, SSL_peek() will return the bytes available in the internal read buffer. If no bytes are buffered in the internal read buffer yet, a call to wolfSSL_peek() will trigger processing of the next record.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **data** buffer where wolfSSL_peek() will place data read.
- **sz** number of bytes to read into data.

See: wolfSSL_read

Return:

- 0 the number of bytes read upon success.
- 0 will be returned upon failure. This may be caused by a either a clean (close notify alert) shutdown or just that the peer closed the connection. Call wolfSSL_get_error() for the specific error code.
- SSL_FATAL_ERROR will be returned upon failure when either an error occurred or, when using nonblocking sockets, the SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE error was received and and the application needs to call wolfSSL_peek() again. Use wolfSSL_get_error() to get a specific error code.
- BAD_FUNC_ARG will be returned when ssl or data is NULL, or sz is negative.

Example

```
WOLFSSL* ssl = 0;
char reply[1024];
...

input = wolfSSL_peek(ssl, reply, sizeof(reply));
if (input > 0) {
    // “input” number of bytes returned into buffer “reply”
}
```

```
int wolfSSL_accept(
    WOLFSSL * ssl
)
```

This function is called on the server side and waits for an SSL client to initiate the SSL/TLS handshake. When this function is called, the underlying communication channel has already been set up. wolfSSL_accept() works with both blocking and non-blocking I/O. When the underlying I/O is non-blocking, wolfSSL_accept() will return when the underlying I/O could not satisfy the needs of wolfSSL_accept to continue the handshake. In this case, a call to wolfSSL_get_error() will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_accept when data is available to read and wolfSSL will pick up where it left off. When using a non_blocking socket, nothing needs to be done, but select() can be used to check for the required condition. If the underlying I/O is blocking, wolfSSL_accept() will only return once the handshake has been finished or an error occurred.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_get_error
- wolfSSL_connect

Return:

- SSL_SUCCESS upon success.
- SSL_FATAL_ERROR will be returned if an error occurred. To get a more detailed error code, call wolfSSL_get_error().

If decrypted application data is waiting to be read, for example early data that has not been read yet or data received during a secure renegotiation, the handshake is not continued and wolfSSL_get_error() returns APP_DATA_READY. Read the data with wolfSSL_read() and then call wolfSSL_accept() again.

Example

```
int ret = 0;
int err = 0;
WOLFSSL* ssl;
char buffer[80];
...

ret = wolfSSL_accept(ssl);
if (ret != SSL_SUCCESS) {
    err = wolfSSL_get_error(ssl, ret);
    printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
}
```

```
int wolfDTLS_accept_stateless(
    WOLFSSL * ssl
)
```

This function statelessly listens for an SSL client to initiate the DTLS handshake. Cookies must be enabled. A cookie-disabled DTLS object is rejected before I/O or callback changes: WOLF-

SSL_FATAL_ERROR is returned and wolfSSL_get_error() reports BAD_STATE_E. Use wolfSSL_accept() for cookie-disabled connections instead.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_accept
- wolfSSL_get_error
- wolfSSL_connect

Return:

- WOLFSSL_SUCCESS ClientHello containing a valid cookie was received. The connection can be continued with wolfSSL_accept().
- WOLFSSL_FAILURE The I/O layer returned WANT_READ. This is either because there is no data to read and we are using non-blocking sockets or we sent a cookie request and we are waiting for a reply. The user should call wolfDTLS_accept_stateless again after data becomes available in the I/O layer.
- WOLFSSL_FATAL_ERROR A fatal error occurred. The ssl object should be free'd and allocated again to continue. wolfSSL_get_error() reports SIDE_ERROR when called on a client object, and BAD_STATE_E when cookies are disabled on this object.

Example

```
int ret = 0;
int err = 0;
WOLFSSL* ssl;
...
do {
    ret = wolfDTLS_accept_stateless(ssl);
    if (ret == WOLFSSL_FATAL_ERROR)
        // re-allocate the ssl object with wolfSSL_free() and wolfSSL_new()
} while (ret != WOLFSSL_SUCCESS);
ret = wolfSSL_accept(ssl);
if (ret != SSL_SUCCESS) {
    err = wolfSSL_get_error(ssl, ret);
    printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
}
```

```
int wolfSSL_send(
    WOLFSSL * ssl,
    const void * data,
    int sz,
    int flags
)
```

This function writes sz bytes from the buffer, data, to the SSL connection, ssl, using the specified flags for the underlying write operation. If necessary wolfSSL_send() will negotiate an SSL/TLS session if the handshake has not already been performed yet by wolfSSL_connect() or wolfSSL_accept(). wolfSSL_send() works with both blocking and non_blocking I/O. When the underlying I/O is non_blocking, wolfSSL_send() will return when the underlying I/O could not satisfy the needs of wolfSSL_send to continue. In this case, a call to wolfSSL_get_error() will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_send() when the underlying I/O is ready. If the underlying I/O is blocking, wolfSSL_send() will only return once the buffer data of size sz has been completely written or an error occurred.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **data** data buffer to send to peer.
- **sz** size, in bytes, of data to be sent to peer.
- **flags** the send flags to use for the underlying send operation.

See:

- wolfSSL_write
- wolfSSL_read
- wolfSSL_recv

Return:

- 0 the number of bytes written upon success.
- 0 will be returned upon failure. Call wolfSSL_get_error() for the specific error code.
- SSL_FATAL_ERROR will be returned upon failure when either an error occurred or, when using non-blocking sockets, the SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE error was received and and the application needs to call wolfSSL_send() again. Use wolfSSL_get_error() to get a specific error code.
- BAD_FUNC_ARG will be returned when ssl or data is NULL, or sz is negative.

Example

```
WOLFSSL* ssl = 0;
char msg[64] = “hello wolfssl!”;
int msgSz = (int)strlen(msg);
int flags = ... ;
...

input = wolfSSL_send(ssl, msg, msgSz, flags);
if (input != msgSz) {
    // wolfSSL_send() failed
}
```

```
int wolfSSL_recv(
    WOLFSSL * ssl,
    void * data,
    int sz,
    int flags
)
```

This function reads sz bytes from the SSL session (ssl) internal read buffer into the buffer data using the specified flags for the underlying recv operation. The bytes read are removed from the internal receive buffer. This function is identical to wolfSSL_read() except that it allows the application to set the recv flags for the underlying read operation. If necessary wolfSSL_recv() will negotiate an SSL/TLS session if the handshake has not already been performed yet by wolfSSL_connect() or wolfSSL_accept(). The SSL/TLS protocol uses SSL records which have a maximum size of 16kB (the max record size can be controlled by the MAX_RECORD_SIZE define in /wolfssl/internal.h). As such, wolfSSL needs to read an entire SSL record internally before it is able to process and decrypt the record. Because of this, a call to wolfSSL_recv() will only be able to return the maximum buffer size which has been decrypted at the time of calling. There may be additional not_yet_decrypted data waiting in the internal wolfSSL receive buffer which will be retrieved and decrypted with the next call to wolfSSL_recv(). If sz is larger than the number of bytes in the internal read buffer, SSL_recv() will return the bytes available in the internal read buffer. If no bytes are buffered in the internal read buffer yet, a call to wolfSSL_recv() will trigger processing of the next record.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **data** buffer where wolfSSL_recv() will place data read.
- **sz** number of bytes to read into data.
- **flags** the recv flags to use for the underlying recv operation.

See:

- wolfSSL_read
- wolfSSL_write
- wolfSSL_peek
- wolfSSL_pending

Return:

- 0 the number of bytes read upon success.
- 0 will be returned upon failure. This may be caused by a either a clean (close notify alert) shutdown or just that the peer closed the connection. Call wolfSSL_get_error() for the specific error code.
- SSL_FATAL_ERROR will be returned upon failure when either an error occurred or, when using non-blocking sockets, the SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE error was received and and the application needs to call wolfSSL_recv() again. Use wolfSSL_get_error() to get a specific error code.
- BAD_FUNC_ARG will be returned when ssl or data is NULL, or sz is negative.

Example

```
WOLFSSL* ssl = 0;
char reply[1024];
int flags = ... ;
...

input = wolfSSL_recv(ssl, reply, sizeof(reply), flags);
if (input > 0) {
    // “input” number of bytes returned into buffer “reply”
}
```

```
int wolfSSL_get_alert_history(
    WOLFSSL * ssl,
    WOLFSSL_ALERT_HISTORY * h
)
```

This function gets the alert history.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **h** a pointer to a WOLFSSL_ALERT_HISTORY structure that will hold the WOLFSSL struct’s alert_history member’s value.

See: wolfSSL_get_error

Return: SSL_SUCCESS returned when the function completed successfully. Either there was alert history or there wasn’t, either way, the return value is SSL_SUCCESS.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(protocol method);
WOLFSSL* ssl = wolfSSL_new(ctx);
WOLFSSL_ALERT_HISTORY* h;
...
wolfSSL_get_alert_history(ssl, h);
// h now has a copy of the ssl->alert_history  contents
```

```
WOLFSSL_SESSION * wolfSSL_get_session(
    WOLFSSL * ssl
)
```

Returns a pointer to the current session (WOLFSSL_SESSION) used in ssl. This is a non-persistent pointer to the WOLFSSL_SESSION object, freed when wolfSSL_free is called, so the call should only be used to inspect or modify the current session. For session resumption use wolfSSL_get1_session(), which returns a reference the caller owns and releases with wolfSSL_SESSION_free().

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See:

- wolfSSL_get1_session
- wolfSSL_set_session

Return:

- pointer If successful the call will return a pointer to the the current SSL session object.
- NULL will be returned if ssl is NULL, the SSL session cache is disabled, wolfSSL doesn't have the Session ID available, or mutex functions fail.

This is the default behaviour: settings.h defines NO_SESSION_CACHE_REF unless WOLFSSL_SESSION_CACHE_REF is defined, so every build selects it without asking. The deprecated behaviour remains available by defining WOLFSSL_SESSION_CACHE_REF, or through –enable-session-cache-ref and -DWOLFSSL_SESSION_CACHE_REF=yes, and such a build warns at compile time, both when building the library and at every call to this function; define WOLFSSL_SESSION_CACHE_REF_WARNED to silence both.

Under WOLFSSL_SESSION_CACHE_REF this function instead returns a persistent session object pointer stored in the local cache. The cache size is finite and there is a risk that the session object will be overwritten by another ssl connection by the time the application calls wolfSSL_set_session() on it. Note the lifetime difference when migrating: the deprecated pointer stayed valid after wolfSSL_free(), the default one does not.

Example

```
WOLFSSL* ssl;
WOLFSSL_SESSION* session;
...
session = wolfSSL_get_session(ssl);
if (session == NULL) {
    // failed to get session pointer
}
...
```

```
void wolfSSL_flush_sessions(
    WOLFSSL_CTX * ctx,
    long tm
)
```

This function flushes session from the session cache which have expired. The time, tm, is used for the time comparison. Note that wolfSSL currently uses a static table for sessions, so no flushing is needed. As such, this function is currently just a stub. This function provides OpenSSL compatibility (SSL_flush_sessions) when wolfSSL is compiled with the OpenSSL compatibility layer.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **tm** time used in session expiration comparison.

See:

- wolfSSL_get1_session
- wolfSSL_set_session

Return: none No returns.

Example

```
WOLFSSL_CTX* ssl;
...
wolfSSL_flush_sessions(ctx, time(0));
```

```
int wolfSSL_GetSessionIndex(
    WOLFSSL * ssl
)
```

This function gets the session index of the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_GetSessionAtIndex

Return: int The function returns an int type representing the sessionIndex within the WOLFSSL struct.

Example

```
WOLFSSL_CTX_new( protocol method );
WOLFSSL* ssl = WOLFSSL_new(ctx);
...
int sesIdx = wolfSSL_GetSessionIndex(ssl);

if(sesIdx < 0 || sesIdx > sizeof(ssl->sessionIndex)/sizeof(int)){
    // You have an out of bounds index number and something is not right.
}
```

```
int wolfSSL_GetSessionAtIndex(
    int index,
    WOLFSSL_SESSION * session
)
```

This function gets the session at specified index of the session cache and copies it into memory. The WOLFSSL_SESSION structure holds the session information. The copy is independent of the cache entry: it does not share the ticket buffer, peer certificate or ex_data with the cache, and it stays valid after the cache entry is overwritten or evicted. The caller owns the copy and releases it with wolfSSL_SESSION_free().

Parameters:

- **index** an int type representing the session index.
- **session** a pointer to a WOLFSSL_SESSION structure to copy into, obtained from wolfSSL_SESSION_new().

See:

- UnLockMutex
- LockMutex
- wolfSSL_GetSessionIndex

Return:

- SSL_SUCCESS returned if the function executed successfully and no errors were thrown.
- BAD_MUTEX_E returned if there was an unlock or lock mutex error.
- SSL_FAILURE returned if the function did not execute successfully.

Example

```
int idx; // The index to locate the session.
WOLFSSL_SESSION* session = wolfSSL_SESSION_new();  // Buffer to copy to.
...
if(wolfSSL_GetSessionAtIndex(idx, session) != SSL_SUCCESS){
    // Failure case.
}
...
wolfSSL_SESSION_free(session);
```

```
WOLFSSL_X509_CHAIN * wolfSSL_SESSION_get_peer_chain(
    WOLFSSL_SESSION * session
)
```

Returns the peer certificate chain from the WOLFSSL_SESSION struct.

Parameters:

- **session** a pointer to a WOLFSSL_SESSION structure.

See:

- wolfSSL_GetSessionAtIndex
- wolfSSL_GetSessionIndex
- AddSession

Return: pointer A pointer to a WOLFSSL_X509_CHAIN structure that contains the peer certification chain.

Example

```
WOLFSSL_SESSION* session;
WOLFSSL_X509_CHAIN* chain;
...
chain = wolfSSL_SESSION_get_peer_chain(session);
if(!chain){
    // There was no chain. Failure case.
}
```

```
int wolfSSL_pending(
    WOLFSSL * ssl
)
```

This function returns the number of bytes which are buffered and available in the SSL object to be read by wolfSSL_read().

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See:

- wolfSSL_recv
- wolfSSL_read
- wolfSSL_peek

Return:

- int This function returns the number of bytes pending.
- SSL_FAILURE will be returned when ssl is NULL.

Example

```
int pending = 0;
WOLFSSL* ssl = 0;
...

pending = wolfSSL_pending(ssl);
printf("There are %d bytes buffered and available for reading", pending);
```

```
int wolfSSL_save_session_cache(
    const char * fname
)
```

This function persists the session cache to file. It doesn't use memsave because of additional memory use.

Parameters:

- **fname** is a constant char pointer that points to a file for writing.

See:

- XFWRITE
- wolfSSL_restore_session_cache
- wolfSSL_memrestore_session_cache

Return:

- SSL_SUCCESS returned if the function executed without error. The session cache has been written to a file.
- SSL_BAD_FILE returned if fname cannot be opened or is otherwise corrupt.
- FWRITE_ERROR returned if XFWRITE failed to write to the file.
- BAD_MUTEX_E returned if there was a mutex lock failure.

Warning: The file holds session master secrets and resumption credentials in the clear. On POSIX systems it is created with mode 0600; on other platforms the permissions are whatever the port's XFOPEN produces, so the caller must place the file where only the intended user can read it. An existing file is additionally tightened with fchmod, which fails on filesystems without permission bits (FAT); define WOLFSSL_NO_FCHMOD to skip that step on such targets.

Example

```
const char* fname;
...
if(wolfSSL_save_session_cache(fname) != SSL_SUCCESS){
    // Fail to write to file.
}
```

```
int wolfSSL_restore_session_cache(
    const char * fname
)
```

This function restores the persistent session cache from file. It does not use memstore because of additional memory use.

Parameters:

- **fname** a constant char pointer file input that will be read.

See:

- XFREAD
- XFOPEN

Return:

- SSL_SUCCESS returned if the function executed without error.

- SSL_BAD_FILE returned if the file passed into the function was corrupted and could not be opened by XFOPEN.
- FREAD_ERROR returned if the file had a read error from XFREAD.
- CACHE_MATCH_ERROR returned if the session cache header match failed.
- BAD_MUTEX_E returned if there was a mutex lock failure.

Example

```
const char *fname;
...
if(wolfSSL_restore_session_cache(fname) != SSL_SUCCESS){
    // Failure case. The function did not return SSL_SUCCESS.
}
```

```
int wolfSSL_memsave_session_cache(
    void * mem,
    int sz
)
```

This function persists session cache to memory.

Parameters:

- **mem** a void pointer representing the destination for the memory copy, XMEMCPY().
- **sz** an int type representing the size of mem.

See:

- XMEMCPY
- wolfSSL_get_session_cache_memsize

Return:

- SSL_SUCCESS returned if the function executed without error. The session cache has been successfully persisted to memory.
- BAD_MUTEX_E returned if there was a mutex lock error.
- BUFFER_E returned if the buffer size was too small.

Example

```
void* mem;
int sz; // Max size of the memory buffer.
…
if(wolfSSL_memsave_session_cache(mem, sz) != SSL_SUCCESS){
    // Failure case, you did not persist the session cache to memory
}
```

```
int wolfSSL_memrestore_session_cache(
    const void * mem,
    int sz
)
```

This function restores the persistent session cache from memory.

Parameters:

- **mem** a constant void pointer containing the source of the restoration.
- **sz** an integer representing the size of the memory buffer.

See: wolfSSL_save_session_cache

Return:

- SSL_SUCCESS returned if the function executed without an error.
- BUFFER_E returned if the memory buffer is too small.
- BAD_MUTEX_E returned if the session cache mutex lock failed.
- CACHE_MATCH_ERROR returned if the session cache header match failed.

Example

```
const void* memoryFile;
int szMf;
...
if(wolfSSL_memrestore_session_cache(memoryFile, szMf) != SSL_SUCCESS){
    // Failure case. SSL_SUCCESS was not returned.
}
```

```
int wolfSSL_get_session_cache_memsize(
    void
)
```

This function returns how large the session cache save buffer should be.

Parameters:

- **none** No parameters.

See: wolfSSL_memrestore_session_cache

Return: int This function returns an integer that represents the size of the session cache save buffer.

Example

```
int sz = // Minimum size for error checking;
...
if(sz < wolfSSL_get_session_cache_memsize()){
    // Memory buffer is too small
}
```

```
int wolfSSL_session_reused(
    WOLFSSL * ssl
)
```

This function returns the resuming member of the options struct. The flag indicates whether or not to reuse a session. If not, a new session must be established.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_SESSION_free
- wolfSSL_GetSessionIndex
- wolfSSL_memsave_session_cache

Return: This function returns an int type held in the Options structure representing the flag for session reuse.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
…
if(!wolfSSL_session_reused(sslResume)){
    // No session reuse allowed.
}
```

```
const char * wolfSSL_get_version(
    WOLFSSL * ssl
)
```

Returns the SSL version being used as a string.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_lib_version

Return:

- “SSLv3” Using SSLv3
- “TLSv1” Using TLSv1
- “TLSv1.1” Using TLSv1.1
- “TLSv1.2” Using TLSv1.2
- “TLSv1.3” Using TLSv1.3
- “DTLS”: Using DTLS

- “DTLSv1.2” Using DTLSv1.2
- “unknown” There was a problem determining which version of TLS being used.

Example

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
WOLFSSL_METHOD method = // Some wolfSSL method
ctx = wolfSSL_CTX_new(method);
ssl = wolfSSL_new(ctx);
printf(wolfSSL_get_version("Using version: %s", ssl));
```

```
int wolfSSL_get_current_cipher_suite(
    WOLFSSL * ssl
)
```

Returns the current cipher suit an ssl session is using.

Parameters:

- **ssl** The SSL session to check.

See:

- wolfSSL_CIPHER_get_name
- wolfSSL_get_current_cipher
- wolfSSL_get_cipher_list

Return:

- ssl->options.cipherSuite An integer representing the current cipher suite.
- 0 The ssl session provided is null.

Example

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
WOLFSSL_METHOD method = // Some wolfSSL method
ctx = wolfSSL_CTX_new(method);
ssl = wolfSSL_new(ctx);

if(wolfSSL_get_current_cipher_suite(ssl) == 0)
{
    // Error getting cipher suite
}
```

```
WOLFSSL_CIPHER * wolfSSL_get_current_cipher(
    WOLFSSL * ssl
)
```

This function returns a pointer to the current cipher in the ssl session. The cipher is filled in from the session on each call, so call it again after the session changes, e.g. after the handshake.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_get_cipher
- wolfSSL_get_cipher_name_internal
- wolfSSL_get_cipher_name

Return:

- The function returns the address of the cipher member of the WOLFSSL struct. This is a pointer to the WOLFSSL_CIPHER structure.
- NULL returned if the WOLFSSL structure is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
…
WOLFSSL_CIPHER* cipherCurr = wolfSSL_get_current_cipher;

if(!cipherCurr){
    // Failure case.
} else {
    // The cipher was returned to cipherCurr
}
```

```
const char * wolfSSL_CIPHER_get_name(
    const WOLFSSL_CIPHER * cipher
)
```

This function matches the cipher suite in the SSL object with the available suites and returns the string representation.

Parameters:

- **cipher** a constant pointer to a WOLFSSL_CIPHER structure.

See:

- wolfSSL_get_cipher
- wolfSSL_get_current_cipher
- wolfSSL_get_cipher_name_internal
- wolfSSL_get_cipher_name

Return:

- string This function returns the string representation of the matched cipher suite.
- none It will return “None” if there are no suites matched.

Example

```
// gets cipher name in the format DHE_RSA ...
const char* wolfSSL_get_cipher_name_internal(WOLFSSL* ssl){
WOLFSSL_CIPHER* cipher;
const char* fullName;
…
cipher = wolfSSL_get_curent_cipher(ssl);
fullName = wolfSSL_CIPHER_get_name(cipher);

if(fullName){
    // sanity check on returned cipher
}
```

```
const char * wolfSSL_get_cipher(
    WOLFSSL *
)
```

This function matches the cipher suite in the SSL object with the available suites.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_CIPHER_get_name
- wolfSSL_get_current_cipher

Return: This function returns the string value of the suite matched. It will return “None” if there are no suites matched.

Example

```
#ifdef WOLFSSL_DTLS
…
// make sure a valid suite is used
if(wolfSSL_get_cipher(ssl) == NULL){
    WOLFSSL_MSG(“Can not match cipher suite imported”);
    return MATCH_SUITE_ERROR;
```

```
}
…
#endif // WOLFSSL_DTLS
```

```
int wolfSSL_BIO_get_mem_data(
    WOLFSSL_BIO * bio,
    void * p
)
```

This is used to set a byte pointer to the start of the internal memory buffer.

Parameters:

- **bio** WOLFSSL_BIO structure to get memory buffer of.
- **p** byte pointer to set to memory buffer.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_set_fp
- wolfSSL_BIO_free

Return:

- size On success the size of the buffer is returned
- SSL_FATAL_ERROR If an error case was encountered.

Example

```
WOLFSSL_BIO* bio;
const byte* p;
int ret;
bio  = wolfSSL_BIO_new(wolfSSL_BIO_s_mem());
ret  = wolfSSL_BIO_get_mem_data(bio, &p);
// check ret value
```

```
void wolfSSL_BIO_set_init(
    WOLFSSL_BIO * bio,
    int init
)
```

This is used to set the init flag of a BIO, indicating whether the BIO has been initialised and is ready for use. Typically called from a custom BIO create callback.

Parameters:

- **bio** WOLFSSL_BIO structure to set the init flag on.

- **init** value to set (0 = not initialised, 1 = initialised).

See:

- wolfSSL_BIO_get_init
- wolfSSL_BIO_new

Example

```
WOLFSSL_BIO* bio;
// inside a custom BIO create callback
wolfSSL_BIO_set_init(bio, 1);
```

```
int wolfSSL_BIO_get_init(
    WOLFSSL_BIO * bio
)
```

This is used to retrieve the init flag of a BIO, indicating whether the BIO has been initialised and is ready for use.

Parameters:

- **bio** WOLFSSL_BIO structure to query.

See:

- wolfSSL_BIO_set_init
- wolfSSL_BIO_new

Return:

- 1 if the BIO has been initialised.
- 0 if the BIO has not been initialised or bio is NULL.

Example

```
WOLFSSL_BIO* bio;
// create bio with custom method
if (wolfSSL_BIO_get_init(bio)) {
    // bio is ready
}
```

```
long wolfSSL_BIO_set_fd(
    WOLFSSL_BIO * b,
    int fd,
    int flag
)
```

Sets the file descriptor for bio to use.

Parameters:

- **b** WOLFSSL_BIO structure to set fd.
- **fd** file descriptor to use.
- **flag** flag for behavior when closing fd.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_free

Return: SSL_SUCCESS(1) upon success.

Example

```
WOLFSSL_BIO* bio;
int fd;
// setup bio
wolfSSL_BIO_set_fd(bio, fd, BIO_NOCLOSE);
```

```
int wolfSSL_BIO_set_close(
    WOLFSSL_BIO * b,
    long flag
)
```

Sets the close flag, used to indicate that the i/o stream should be closed when the BIO is freed.

Parameters:

- **b** WOLFSSL_BIO structure.
- **flag** flag for behavior when closing i/o stream.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_free

Return: SSL_SUCCESS(1) upon success.

Example

```
WOLFSSL_BIO* bio;
// setup bio
wolfSSL_BIO_set_close(bio, BIO_NOCLOSE);
```

```
WOLFSSL_BIO_METHOD * wolfSSL_BIO_s_socket(
    void
)
```

This is used to get a BIO_SOCKET type WOLFSSL_BIO_METHOD.

Parameters:

- **none** No parameters.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem

Return: WOLFSSL_BIO_METHOD pointer to a WOLFSSL_BIO_METHOD structure that is a socket type

Example

```
WOLFSSL_BIO* bio;
bio = wolfSSL_BIO_new(wolfSSL_BIO_s_socket);
```

```
int wolfSSL_BIO_set_write_buf_size(
    WOLFSSL_BIO * b,
    long size
)
```

This is used to set the size of write buffer for a WOLFSSL_BIO. If write buffer has been previously set this function will free it when resetting the size. It is similar to wolfSSL_BIO_reset in that it resets read and write indexes to 0.

Parameters:

- **b** WOLFSSL_BIO structure to set write buffer size.
- **size** size of buffer to allocate.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_free

Return:

- SSL_SUCCESS On successfully setting the write buffer.
- SSL_FAILURE If an error case was encountered.

Example

```
WOLFSSL_BIO* bio;
int ret;
bio = wolfSSL_BIO_new(wolfSSL_BIO_s_mem());
ret = wolfSSL_BIO_set_write_buf_size(bio, 15000);
// check return value
```

```
int wolfSSL_BIO_make_bio_pair(
    WOLFSSL_BIO * b1,
    WOLFSSL_BIO * b2
)
```

This is used to pair two bios together. A pair of bios acts similar to a two way pipe writing to one can be read by the other and vice versa. It is expected that both bios be in the same thread, this function is not thread safe. Freeing one of the two bios removes both from being paired. If a write buffer size was not previously set for either of the bios it is set to a default size of 17000 (WOLFSSL_BIO_SIZE) before being paired.

Parameters:

- **b1** WOLFSSL_BIO structure to set pair.
- **b2** second WOLFSSL_BIO structure to complete pair.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_free

Return:

- SSL_SUCCESS On successfully pairing the two bios.
- SSL_FAILURE If an error case was encountered.

Example

```
WOLFSSL_BIO* bio;
WOLFSSL_BIO* bio2;
int ret;
bio  = wolfSSL_BIO_new(wolfSSL_BIO_s_bio());
bio2 = wolfSSL_BIO_new(wolfSSL_BIO_s_bio());
ret = wolfSSL_BIO_make_bio_pair(bio, bio2);
// check ret value
```

```
int wolfSSL_BIO_ctrl_reset_read_request(
    WOLFSSL_BIO * b
)
```

This is used to set the read request flag back to 0.

Parameters:

- **b** WOLFSSL_BIO structure to set read request flag.

See:

- wolfSSL_BIO_new, wolfSSL_BIO_s_mem
- wolfSSL_BIO_new, wolfSSL_BIO_free

Return:

- SSL_SUCCESS On successfully setting value.
- SSL_FAILURE If an error case was encountered.

Example

```
WOLFSSL_BIO* bio;
int ret;
...
ret = wolfSSL_BIO_ctrl_reset_read_request(bio);
// check ret value
```

```
int wolfSSL_BIO_nread0(
    WOLFSSL_BIO * bio,
    char ** buf
)
```

This is used to get a buffer pointer for reading from. Unlike wolfSSL_BIO_nread the internal read index is not advanced by the number returned from the function call. Reading past the value returned can result in reading out of array bounds.

Parameters:

- **bio** WOLFSSL_BIO structure to read from.
- **buf** pointer to set at beginning of read array.

See: wolfSSL_BIO_new

Return: >=0 on success return the number of bytes to read

Example

```
WOLFSSL_BIO* bio;
char* bufPt;
int ret;
// set up bio
ret = wolfSSL_BIO_nread0(bio, &bufPt); // read as many bytes as possible
```

```
// handle negative ret check
// read ret bytes from bufPt



int wolfSSL_BIO_nread(
    WOLFSSL_BIO * bio,
    char ** buf,
    int num
)
```

This is used to get a buffer pointer for reading from. The internal read index is advanced by the number returned from the function call with buf being pointed to the beginning of the buffer to read from. In the case that less bytes are in the read buffer than the value requested with num the lesser value is returned. Reading past the value returned can result in reading out of array bounds.

Parameters:

- **bio** WOLFSSL_BIO structure to read from.
- **buf** pointer to set at beginning of read array.
- **num** number of bytes to try and read.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_nwrite

Return:

- =0 on success return the number of bytes to read
- WOLFSSL_BIO_ERROR(-1) on error case with nothing to read return -1

Example

```
WOLFSSL_BIO* bio;
char* bufPt;
int ret;

// set up bio
ret = wolfSSL_BIO_nread(bio, &bufPt, 10); // try to read 10 bytes
// handle negative ret check
// read ret bytes from bufPt



int wolfSSL_BIO_nwrite(
    WOLFSSL_BIO * bio,
    char ** buf,
    int num
)
```

Gets a pointer to the buffer for writing as many bytes as returned by the function. Writing more bytes to the pointer returned then the value returned can result in writing out of bounds.

Parameters:

- **bio** WOLFSSL_BIO structure to write to.
- **buf** pointer to buffer to write to.
- **num** number of bytes desired to be written.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_free
- wolfSSL_BIO_nread

Return:

- int Returns the number of bytes that can be written to the buffer pointer returned.
- WOLFSSL_BIO_UNSET(-2) in the case that is not part of a bio pair
- WOLFSSL_BIO_ERROR(-1) in the case that there is no more room to write to

Example

```
WOLFSSL_BIO* bio;
char* bufPt;
int ret;
// set up bio
ret = wolfSSL_BIO_nwrite(bio, &bufPt, 10); // try to write 10 bytes
// handle negative ret check
// write ret bytes to bufPt
```

```
int wolfSSL_BIO_reset(
    WOLFSSL_BIO * bio
)
```

Resets bio to an initial state. As an example for type BIO_BIO this resets the read and write index.

Parameters:

- **bio** WOLFSSL_BIO structure to reset.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_free

Return:

- 0 On successfully resetting the bio.

- WOLFSSL_BIO_ERROR(-1) Returned on bad input or unsuccessful reset.

Example

```
WOLFSSL_BIO* bio;
// setup bio
wolfSSL_BIO_reset(bio);
//use pt
```

```
int wolfSSL_BIO_seek(
    WOLFSSL_BIO * bio,
    int ofs
)
```

This function adjusts the file pointer to the offset given. This is the offset from the head of the file.

Parameters:

- **bio** WOLFSSL_BIO structure to set.
- **ofs** offset into file.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_set_fp
- wolfSSL_BIO_free

Return:

- 0 On successfully seeking.
- -1 If an error case was encountered.

Example

```
WOLFSSL_BIO* bio;
XFILE fp;
int ret;
bio  = wolfSSL_BIO_new(wolfSSL_BIO_s_file());
ret  = wolfSSL_BIO_set_fp(bio, &fp);
// check ret value
ret  = wolfSSL_BIO_seek(bio, 3);
// check ret value
```

```
int wolfSSL_BIO_write_filename(
    WOLFSSL_BIO * bio,
    char * name
)
```

This is used to set and write to a file. WIll overwrite any data currently in the file and is set to close the file when the bio is freed.

Parameters:

- **bio** WOLFSSL_BIO structure to set file.
- **name** name of file to write to.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_file
- wolfSSL_BIO_set_fp
- wolfSSL_BIO_free

Return:

- SSL_SUCCESS On successfully opening and setting file.
- SSL_FAILURE If an error case was encountered.

Example

```
WOLFSSL_BIO* bio;
int ret;
bio  = wolfSSL_BIO_new(wolfSSL_BIO_s_file());
ret  = wolfSSL_BIO_write_filename(bio, “test.txt”);
// check ret value
```

```
long wolfSSL_BIO_set_mem_eof_return(
    WOLFSSL_BIO * bio,
    int v
)
```

This is used to set the end of file value. Common value is -1 so as not to get confused with expected positive values.

Parameters:

- **bio** WOLFSSL_BIO structure to set end of file value.
- **v** value to set in bio.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_set_fp
- wolfSSL_BIO_free

Return: 0 returned on completion

Example

```
WOLFSSL_BIO* bio;
int ret;
bio  = wolfSSL_BIO_new(wolfSSL_BIO_s_mem());
ret  = wolfSSL_BIO_set_mem_eof_return(bio, -1);
// check ret value
```

```
long wolfSSL_BIO_get_mem_ptr(
    WOLFSSL_BIO * bio,
    WOLFSSL_BUF_MEM ** m
)
```

This is a getter function for WOLFSSL_BIO memory pointer.

Parameters:

- **bio** pointer to the WOLFSSL_BIO structure for getting memory pointer.
- **m** pointer to WOLFSSL_BUF_MEM structure. Is set to point to bio's memory.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem

Return:

- SSL_SUCCESS On successfully getting the pointer SSL_SUCCESS is returned (currently value of 1).
- SSL_FAILURE Returned if NULL arguments are passed in (currently value of 0).

Example

```
WOLFSSL_BIO* bio;
WOLFSSL_BUF_MEM* pt;
// setup bio
wolfSSL_BIO_get_mem_ptr(bio, &pt);
//use pt
```

```
const char * wolfSSL_lib_version(
    void
)
```

This function returns the current library version.

Parameters:

- **none** No parameters.

See: word32_wolfSSL_lib_version_hex

Return: LIBWOLFSSL_VERSION_STRING a const char pointer defining the version.

Example

```
char version[MAXSIZE];
version = wolfSSL_KeepArrays();
…
if(version != ExpectedVersion){
    // Handle the mismatch case
}
```

```
word32 wolfSSL_lib_version_hex(
    void
)
```

This function returns the current library version in hexadecimal notation.

Parameters:

- **none** No parameters.

See: wolfSSL_lib_version

Return: LILBWOLFSSL_VERSION_HEX returns the hexadecimal version defined in wolfssl/version.h.

Example

```
word32 libV;
libV = wolfSSL_lib_version_hex();

if(libV != EXPECTED_HEX){
    // How to handle an unexpected value
} else {
    // The expected result for libV
}
```

```
int wolfSSL_negotiate(
    WOLFSSL * ssl
)
```

Performs the actual connect or accept based on the side of the SSL method. If called from the client side then an wolfSSL_connect() is done while a wolfSSL_accept() is performed if called from the server side.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See:

- SSL_connect
- SSL_accept

Return:

- SSL_SUCCESS will be returned if successful. (Note, older versions will return 0.)
- SSL_FATAL_ERROR will be returned if the underlying call resulted in an error. Use wolfSSL_get_error() to get a specific error code.

If decrypted application data is waiting to be read, the handshake is not continued and wolfSSL_get_error() returns APP_DATA_READY. Read the data with wolfSSL_read() and then call wolfSSL_negotiate() again.

Example

```
int ret = SSL_FATAL_ERROR;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_negotiate(ssl);
if (ret == SSL_FATAL_ERROR) {
    // SSL establishment failed
int error_code = wolfSSL_get_error(ssl);
...
}
...
```

```
int wolfSSL_connect_cert(
    WOLFSSL * ssl
)
```

This function is called on the client side and initiates an SSL/TLS handshake with a server only long enough to get the peer's certificate chain. When this function is called, the underlying communication channel has already been set up. wolfSSL_connect_cert() works with both blocking and non-blocking I/O. When the underlying I/O is non-blocking, wolfSSL_connect_cert() will return when the underlying I/O could not satisfy the needs of wolfSSL_connect_cert() to continue the handshake. In this case, a call to wolfSSL_get_error() will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_connect_cert() when the underlying I/O is ready and wolfSSL will pick up where it left off. When using a non_blocking socket, nothing needs to be done, but select() can be used to check for the required condition. If the underlying I/O is blocking, wolfSSL_connect_cert() will only return once the peer's certificate chain has been received.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_get_error
- wolfSSL_connect
- wolfSSL_accept

Return:

- SSL_SUCCESS upon success.
- SSL_FAILURE will be returned if the SSL session parameter is NULL.
- SSL_FATAL_ERROR will be returned if an error occurred. To get a more detailed error code, call wolfSSL_get_error().

Example

```
int ret = 0;
int err = 0;
WOLFSSL* ssl;
char buffer[80];
...
ret = wolfSSL_connect_cert(ssl);
if (ret != SSL_SUCCESS) {
    err = wolfSSL_get_error(ssl, ret);
    printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
}
```

```
int wolfSSL_writev(
    WOLFSSL * ssl,
    const struct iovec * iov,
    int iovcnt
)
```

Simulates writev semantics but doesn't actually do block at a time because of SSL_write() behavior and because front adds may be small. Makes porting into software that uses writev easier.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **iov** array of I/O vectors to write
- **iovcnt** number of vectors in iov array.

See: wolfSSL_write

Return:

- 0 the number of bytes written upon success.
- 0 will be returned upon failure. Call wolfSSL_get_error() for the specific error code.
- MEMORY_ERROR will be returned if a memory error was encountered.
- BAD_FUNC_ARG will be returned when ssl is NULL, iovcnt is negative, or iov is NULL with a non-zero iovcnt.

- SSL_FATAL_ERROR will be returned upon failure when either an error occurred or, when using non-blocking sockets, the SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE error was received and and the application needs to call wolfSSL_write() again. Use wolfSSL_get_error() to get a specific error code.
- BUFFER_E will be returned when the total length of the segments overflows.

Example

```
WOLFSSL* ssl = 0;
char *bufA = “hello\n”;
char *bufB = “hello world\n”;
int iovcnt;
struct iovec iov[2];

iov[0].iov_base = buffA;
iov[0].iov_len = strlen(buffA);
iov[1].iov_base = buffB;
iov[1].iov_len = strlen(buffB);
iovcnt = 2;
...
ret = wolfSSL_writev(ssl, iov, iovcnt);
// wrote “ret” bytes, or error if <= 0.
```

```
unsigned char wolfSSL_SNI_Status(
    WOLFSSL * ssl,
    unsigned char type
)
```

This function gets the status of an SNI object.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **type** the SNI type.

See:

- TLSX_SNI_Status
- TLSX_SNI_find
- TLSX_Find

Return:

- value This function returns the byte value of the SNI struct’s status member if the SNI is not NULL.
- 0 if the SNI object is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
…
#define AssertIntEQ(x, y) AssertInt(x, y, ==, !=)
…
Byte type = WOLFSSL_SNI_HOST_NAME;
char* request = (char*)&type;
AssertIntEQ(WOLFSSL_SNI_NO_MATCH, wolfSSL_SNI_Status(ssl, type));
…
```

```
int wolfSSL_UseSecureRenegotiation(
    WOLFSSL * ssl
)
```

This function forces secure renegotiation for the supplied WOLFSSL structure. This is not recommended.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- TLSX_Find
- TLSX_UseSecureRenegotiation

Return:

- SSL_SUCCESS Successfully set secure renegotiation.
- BAD_FUNC_ARG Returns error if ssl is null.
- MEMORY_E Returns error if unable to allocate memory for secure renegotiation.

Example

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
WOLFSSL_METHOD method = // Some wolfSSL method
ctx = wolfSSL_CTX_new(method);
ssl = wolfSSL_new(ctx);

if(wolfSSL_UseSecureRenegotiation(ssl) != SSL_SUCCESS)
{
    // Error setting secure renegotiation
}
```

```
int wolfSSL_Rehandshake(
    WOLFSSL * ssl
)
```

This function executes a secure renegotiation handshake; this is user forced as wolfSSL discourages this functionality.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_negotiate
- wc_InitSha512
- wc_InitSha384
- wc_InitSha256
- wc_InitSha
- wc_InitMd5

Return:

- SSL_SUCCESS returned if the function executed without error.
- BAD_FUNC_ARG returned if an unacceptable argument was passed in a subroutine.
- SECURE_RENEGOTIATION_E returned if there was an error with renegotiating the handshake.
- SSL_FATAL_ERROR returned if there was an error with the server or client configuration and the renegotiation could not be completed. See wolfSSL_negotiate().
- SSL_FAILURE will be returned when ssl is NULL.

If decrypted application data is waiting to be read, the renegotiation is not continued and wolfSSL_get_error() returns APP_DATA_READY. Read the data with wolfSSL_read(), which also moves the renegotiation forward, or call wolfSSL_Rehandshake() again once it has been read.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_Rehandshake(ssl) != SSL_SUCCESS){
    // There was an error and the rehandshake is not successful.
}
```

```
int wolfSSL_UseSessionTicket(
    WOLFSSL * ssl
)
```

Force provided WOLFSSL structure to use session ticket. The constant HAVE_SESSION_TICKET should be defined and the constant NO_WOLFSSL_CLIENT should not be defined to use this function.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: TLSX_UseSessionTicket

Return:

- SSL_SUCCESS Successfully set use session ticket.
- BAD_FUNC_ARG Returned if ssl is null.
- MEMORY_E Error allocating memory for setting session ticket.

Example

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
WOLFSSL_METHOD method = // Some wolfSSL method
ctx = wolfSSL_CTX_new(method);
ssl = wolfSSL_new(ctx);

if(wolfSSL_UseSessionTicket(ssl) != SSL_SUCCESS)
{
    // Error setting session ticket
}
```

```
int wolfSSL_get_SessionTicket(
    WOLFSSL * ssl,
    unsigned char * buf,
    word32 * bufSz
)
```

This function copies the ticket member of the Session structure to the buffer. If buf is NULL and bufSz is non-NULL, bufSz will be set to the ticket length.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **buf** a byte pointer representing the memory buffer.
- **bufSz** a word32 pointer representing the buffer size.

See:

- wolfSSL_UseSessionTicket
- wolfSSL_set_SessionTicket

Return:

- SSL_SUCCESS returned if the function executed without error.
- BAD_FUNC_ARG returned if ssl or bufSz is NULL, or if bufSz is non-NULL and buf is NULL
- LENGTH_ONLY_E will be returned when buf is NULL, with the required length returned through bufSz.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
byte* buf;
word32 bufSz;  // Initialize with buf size
…
if(wolfSSL_get_SessionTicket(ssl, buf, bufSz) <= 0){
    // Nothing was written to the buffer
} else {
    // the buffer holds the content from ssl->session->ticket
}
```

```
int wolfSSL_set_SessionTicket(
    WOLFSSL * ssl,
    const unsigned char * buf,
    word32 bufSz
)
```

This function sets the ticket member of the WOLFSSL_SESSION structure within the WOLFSSL struct. The buffer passed into the function is copied to memory.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **buf** a byte pointer that gets loaded into the ticket member of the session structure.
- **bufSz** a word32 type that represents the size of the buffer.

See: wolfSSL_set_SessionTicket_cb

Return:

- SSL_SUCCESS returned on successful execution of the function. The function returned without errors.
- BAD_FUNC_ARG returned if the WOLFSSL structure is NULL. This will also be thrown if the buf argument is NULL but the bufSz argument is not zero.
- MEMORY_ERROR will be returned when the ticket cannot be allocated.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
byte* buffer; // File to load
word32 bufSz;
...
if(wolfSSL_KeepArrays(ssl, buffer, bufSz) != SSL_SUCCESS){
    // There was an error loading the buffer to memory.
}
```

```
int wolfSSL_PrintSessionStats(
    void
)
```

This function prints the statistics from the session.

Parameters:

- **none** No parameters.

See: wolfSSL_get_session_stats

Return:

- SSL_SUCCESS returned if the function and subroutines return without error. The session stats have been successfully retrieved and printed.
- BAD_FUNC_ARG returned if the subroutine wolfSSL_get_session_stats() was passed an unacceptable argument.
- BAD_MUTEX_E returned if there was a mutex error in the subroutine.

Example

```
// You will need to have a session object to retrieve stats from.
if(wolfSSL_PrintSessionStats(void) != SSL_SUCCESS   ){
    // Did not print session stats
}
```

```
int wolfSSL_get_session_stats(
    unsigned int * active,
    unsigned int * total,
    unsigned int * peak,
    unsigned int * maxSessions
)
```

This function gets the statistics for the session.

Parameters:

- **active** a word32 pointer representing the total current sessions.
- **total** a word32 pointer representing the total sessions.
- **peak** a word32 pointer representing the peak sessions.
- **maxSessions** a word32 pointer representing the maximum sessions.

See: wolfSSL_PrintSessionStats

Return:

- SSL_SUCCESS returned if the function and subroutines return without error. The session stats have been successfully retrieved and printed.
- BAD_FUNC_ARG returned if the subroutine wolfSSL_get_session_stats() was passed an unacceptable argument.

- BAD_MUTEX_E returned if there was a mutex error in the subroutine.

Example

```
int wolfSSL_PrintSessionStats(void){
…
ret = wolfSSL_get_session_stats(&totalSessionsNow,
&totalSessionsSeen, &peak, &maxSessions);
…
return ret;
```

```
long wolfSSL_BIO_set_fp(
    WOLFSSL_BIO * bio,
    XFILE fp,
    int c
)
```

This is used to set the internal file pointer for a BIO.

Parameters:

- **bio** WOLFSSL_BIO structure to set pair.
- **fp** file pointer to set in bio.
- **c** close file behavior flag.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_get_fp
- wolfSSL_BIO_free

Return:

- SSL_SUCCESS On successfully setting file pointer.
- SSL_FAILURE If an error case was encountered.

Example

```
WOLFSSL_BIO* bio;
XFILE fp;
int ret;
bio  = wolfSSL_BIO_new(wolfSSL_BIO_s_file());
ret  = wolfSSL_BIO_set_fp(bio, fp, BIO_CLOSE);
// check ret value
```

```
long wolfSSL_BIO_get_fp(
    WOLFSSL_BIO * bio,
    XFILE * fp
)
```

This is used to get the internal file pointer for a BIO.

Parameters:

- **bio** WOLFSSL_BIO structure to set pair.
- **fp** file pointer to set in bio.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_set_fp
- wolfSSL_BIO_free

Return:

- SSL_SUCCESS On successfully getting file pointer.
- SSL_FAILURE If an error case was encountered.

Example

```
WOLFSSL_BIO* bio;
XFILE fp;
int ret;
bio  = wolfSSL_BIO_new(wolfSSL_BIO_s_file());
ret  = wolfSSL_BIO_get_fp(bio, &fp);
// check ret value
```

```
size_t wolfSSL_BIO_ctrl_pending(
    WOLFSSL_BIO * b
)
```

Gets the number of pending bytes to read. If BIO type is BIO_BIO then is the number to read from pair. If BIO contains an SSL object then is pending data from SSL object (wolfSSL_pending(ssl)). If is BIO_MEMORY type then returns the size of memory buffer.

Parameters:

- **bio** pointer to the WOLFSSL_BIO structure that has already been created.

See:

- wolfSSL_BIO_make_bio_pair
- wolfSSL_BIO_new

Return: >=0 number of pending bytes.

Example

```
WOLFSSL_BIO* bio;
int pending;
bio = wolfSSL_BIO_new();
…
pending = wolfSSL_BIO_ctrl_pending(bio);
```

```
int wolfSSL_set_jobject(
    WOLFSSL * ssl,
    void * objPtr
)
```

This function sets the jObjectRef member of the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **objPtr** a void pointer that will be set to jObjectRef.

See: wolfSSL_get_jobject

Return:

- SSL_SUCCESS returned if jObjectRef is properly set to objPtr.
- SSL_FAILURE returned if the function did not properly execute and jObjectRef is not set.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = WOLFSSL_new();
void* objPtr = &obj;
...
if(wolfSSL_set_jobject(ssl, objPtr)){
    // The success case
}
```

```
void * wolfSSL_get_jobject(
    WOLFSSL * ssl
)
```

This function returns the jObjectRef member of the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_set_jobject

Return:

- value If the WOLFSSL struct is not NULL, the function returns the jObjectRef value.
- NULL returned if the WOLFSSL struct is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL(ctx);
...
void* jobject = wolfSSL_get_jobject(ssl);

if(jobject != NULL){
    // Success case
}
```

```
int wolfSSL_connect(
    WOLFSSL * ssl
)
```

This function is called on the client side and initiates an SSL/TLS handshake with a server. When this function is called, the underlying communication channel has already been set up. wolfSSL_connect() works with both blocking and non-blocking I/O. When the underlying I/O is non-blocking, wolfSSL_connect() will return when the underlying I/O could not satisfy the needs of wolfSSL_connect to continue the handshake. In this case, a call to wolfSSL_get_error() will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_connect() when the underlying I/O is ready and wolfSSL will pick up where it left off. When using a non_blocking socket, nothing needs to be done, but select() can be used to check for the required condition. If the underlying I/O is blocking, wolfSSL_connect() will only return once the handshake has been finished or an error occurred. wolfSSL takes a different approach to certificate verification than OpenSSL does. The default policy for the client is to verify the server, this means that if you don't load CAs to verify the server you'll get a connect error, unable to verify (_155). It you want to mimic OpenSSL behavior of having SSL_connect succeed even if verifying the server fails and reducing security you can do this by calling: SSL_CTX_set_verify(ctx, SSL_VERIFY_NONE, 0); before calling SSL_new(); Though it's not recommended.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_get_error
- wolfSSL_accept

Return:

- SSL_SUCCESS If successful.

- SSL_FATAL_ERROR will be returned if an error occurred. To get a more detailed error code, call wolfSSL_get_error().
- BAD_FUNC_ARG will be returned when ssl is NULL.

If decrypted application data is waiting to be read, for example data received during a secure renegotiation, the handshake is not continued and wolfSSL_get_error() returns APP_DATA_READY. Read the data with wolfSSL_read() and then call wolfSSL_connect() again.

Example

```
int ret = 0;
int err = 0;
WOLFSSL* ssl;
char buffer[80];
...
ret = wolfSSL_connect(ssl);
if (ret != SSL_SUCCESS) {
err = wolfSSL_get_error(ssl, ret);
printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
}
```

```
int wolfSSL_update_keys(
    WOLFSSL * ssl
)
```

This function is called on a TLS v1.3 client or server wolfSSL to force the rollover of keys. A KeyUpdate message is sent to the peer and new keys are calculated for encryption. The peer will send back a KeyUpdate message and the new decryption keys will then be calculated. This function can only be called after a handshake has been completed.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_write

Return:

- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3.
- WANT_WRITE if the writing is not ready.
- BAD_STATE_E if the connection has already performed the maximum number of key updates. RFC 9846, Section 4.7.3 caps a TLS 1.3 sender at 2^48-1 key updates; beyond that the connection must be closed rather than rekeyed. Note that a KeyUpdate arriving from the peer with request_update set is ignored once this cap is reached, rather than failing the connection, so only an application-initiated update reports this error.
- WOLFSSL_SUCCESS if successful.

Example

```
int ret;
WOLFSSL* ssl;
```

```
...
ret = wolfSSL_update_keys(ssl);
if (ret == WANT_WRITE) {
    // need to call again when I/O ready
}
else if (ret != WOLFSSL_SUCCESS) {
    // failed to send key update
}
```

```
int wolfSSL_key_update_response(
    WOLFSSL * ssl,
    int * required
)
```

This function is called on a TLS v1.3 client or server wolfSSL to determine whether a rollover of keys is in progress. When wolfSSL_update_keys() is called, a KeyUpdate message is sent and the encryption key is updated. The decryption key is updated when the response is received.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **required** 0 when no key update response is required. 1 when a key update response from the peer is still outstanding.

See: wolfSSL_update_keys

Return:

- 0 on successful.
- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3.

Example

```
int ret;
WOLFSSL* ssl;
int required;
...
ret = wolfSSL_key_update_response(ssl, &required);
if (ret != 0) {
    // bad parameters
}
if (required) {
    // encrypt Key updated, awaiting response to change decrypt key
}
```

```
int wolfSSL_request_certificate(
    WOLFSSL * ssl
)
```

This function requests a client certificate from the TLS v1.3 client. This is useful when a web server is serving some pages that require client authentication and others that don't. A maximum of 256 requests can be sent on a connection.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_allow_post_handshake_auth
- wolfSSL_write

Return:

- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3.
- WANT_WRITE if the writing is not ready.
- SIDE_ERROR if called with a client.
- NOT_READY_ERROR if called when the handshake is not finished.
- POST_HAND_AUTH_ERROR if posthandshake authentication is disallowed.
- MEMORY_E if dynamic memory allocation fails.
- WOLFSSL_SUCCESS if successful.

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_request_certificate(ssl);
if (ret == WANT_WRITE) {
    // need to call again when I/O ready
}
else if (ret != WOLFSSL_SUCCESS) {
    // failed to request a client certificate
}
```

```
int wolfSSL_connect_TLSv13(
    WOLFSSL * ssl
)
```

This function is called on the client side and initiates a TLS v1.3 handshake with a server. When this function is called, the underlying communication channel has already been set up. wolfSSL_connect() works with both blocking and non_blocking I/O. When the underlying I/O is non_blocking, wolfSSL_connect() will return when the underlying I/O could not satisfy the needs of wolfSSL_connect to continue the handshake. In this case, a call to wolfSSL_get_error() will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_connect() when the underlying I/O is ready and wolfSSL will pick up where it left off. When using a non_blocking socket, nothing needs to be done, but select() can be used to check for the required condition. If the underlying I/O is blocking, wolfSSL_connect() will only return once the handshake has been finished or an error occurred. wolfSSL takes a different approach to certificate

verification than OpenSSL does. The default policy for the client is to verify the server, this means that if you don't load CAs to verify the server you'll get a connect error, unable to verify (_155). It you want to mimic OpenSSL behavior of having SSL_connect succeed even if verifying the server fails and reducing security you can do this by calling: SSL_CTX_set_verify(ctx, SSL_VERIFY_NONE, 0); before calling SSL_new(); Though it's not recommended.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_get_error
- wolfSSL_connect
- wolfSSL_accept_TLSv13
- wolfSSL_accept

Return:

- SSL_SUCCESS upon success.
- SSL_FATAL_ERROR will be returned if an error occurred. To get a more detailed error code, call wolfSSL_get_error().

If decrypted application data is waiting to be read, the handshake is not continued and wolfSSL_get_error() returns APP_DATA_READY. Read the data with wolfSSL_read() and then call wolfSSL_connect_TLSv13() again.

Example

```
int ret = 0;
int err = 0;
WOLFSSL* ssl;
char buffer[80];
...

ret = wolfSSL_connect_TLSv13(ssl);
if (ret != SSL_SUCCESS) {
    err = wolfSSL_get_error(ssl, ret);
    printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
}
```

```
wolfSSL_accept_TLSv13(
    WOLFSSL * ssl
)
```

This function is called on the server side and waits for a SSL/TLS client to initiate the SSL/TLS handshake. When this function is called, the underlying communication channel has already been set up. wolfSSL_accept() works with both blocking and non_blocking I/O. When the underlying I/O is non_blocking, wolfSSL_accept() will return when the underlying I/O could not satisfy the needs of wolfSSL_accept to continue the handshake. In this case, a call to wolfSSL_get_error() will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call

to wolfSSL_accept when data is available to read and wolfSSL will pick up where it left off. When using a non_blocking socket, nothing needs to be done, but select() can be used to check for the required condition. If the underlying I/O is blocking, wolfSSL_accept() will only return once the handshake has been finished or an error occurred. Call this function when expecting a TLS v1.3 connection though older version ClientHello messages are supported.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_get_error
- wolfSSL_connect_TLSv13
- wolfSSL_connect
- wolfSSL_accept_TLSv13
- wolfSSL_accept

Return:

- SSL_SUCCESS upon success.
- SSL_FATAL_ERROR will be returned if an error occurred. To get a more detailed error code, call wolfSSL_get_error().

Example

```
int ret = 0;
int err = 0;
WOLFSSL* ssl;
char buffer[80];
...

ret = wolfSSL_accept_TLSv13(ssl);
if (ret != SSL_SUCCESS) {
    err = wolfSSL_get_error(ssl, ret);
    printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
}
```

```
int wolfSSL_write_early_data(
    WOLFSSL * ssl,
    const void * data,
    int sz,
    int * outSz
)
```

This function writes early data to the server on resumption. Call this function before wolfSSL_connect() or wolfSSL_connect_TLSv13(). This function is only used with clients.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **data** the buffer holding the early data to write to server.
- **sz** the amount of early data to write in bytes.
- **outSz** the amount of early data written in bytes.

See:

- wolfSSL_read_early_data
- wolfSSL_connect
- wolfSSL_connect_TLSv13

Return:

- BAD_FUNC_ARG if a pointer parameter is NULL, sz is less than 0 or not using TLSv1.3.
- SIDE_ERROR if called with a server.
- BAD_STATE_E if invoked without a valid session or without a valid PSK cb
- WOLFSSL_FATAL_ERROR if the connection is not made, or if the AEAD key usage limit would be exceeded by this write, in which case wolfSSL_get_error() reports TOO_MUCH_EARLY_DATA. A KeyUpdate cannot be performed while sending early data (RFC 9846, Section 5.5), so no further early data can be sent on this connection. The write is all-or-nothing as usual: a value less than sz is never returned. Note that a failed call may still have put records on the wire before the limit was reached, so treat the connection as unusable for early data rather than resuming the send from an offset.
- the amount of early data written in bytes if successful.

Example

```
int ret = 0;
int err = 0;
WOLFSSL* ssl;
byte earlyData[] = { early data };
int outSz;
char buffer[80];
...

ret = wolfSSL_write_early_data(ssl, earlyData, sizeof(earlyData), &outSz);
if (ret < 0) {
    err = wolfSSL_get_error(ssl, ret);
    printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
    goto err_label;
}
if (outSz < sizeof(earlyData)) {
    // not all early data was sent
}
ret = wolfSSL_connect_TLSv13(ssl);
if (ret != SSL_SUCCESS) {
    err = wolfSSL_get_error(ssl, ret);
    printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
}
```

```
int wolfSSL_read_early_data(
```

```
    WOLFSSL * ssl,
    void * data,
    int sz,
    int * outSz
)
```

This function reads any early data from a client on resumption. Call this function instead of wolfSSL_accept() or wolfSSL_accept_TLSv13() to accept a client and read any early data in the handshake. The function should be invoked until wolfSSL_is_init_finished() returns true. Early data may be sent by the client in multiple messages. If there is no early data then the handshake will be processed as normal. This function is only used with servers.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **data** a buffer to hold the early data read from client.
- **sz** size of the buffer in bytes.
- **outSz** number of bytes of early data read.

See:

- wolfSSL_write_early_data
- wolfSSL_accept
- wolfSSL_accept_TLSv13

Return:

- BAD_FUNC_ARG if a pointer parameter is NULL, sz is less than 0 or not using TLSv1.3.
- SIDE_ERROR if called with a client.
- WOLFSSL_FATAL_ERROR if accepting a connection fails.
- Number of early data bytes read (may be zero).

Example

```
int ret = 0;
int err = 0;
WOLFSSL* ssl;
byte earlyData[128];
int outSz;
char buffer[80];
...

do {
    ret = wolfSSL_read_early_data(ssl, earlyData, sizeof(earlyData), &outSz);
    if (ret < 0) {
        err = wolfSSL_get_error(ssl, ret);
        printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
    }
    if (outSz > 0) {
        // early data available
    }
} while (!wolfSSL_is_init_finished(ssl));
```

```
int wolfSSL_inject(
    WOLFSSL * ssl,
    const void * data,
    int sz
)
```

This function is called to inject data into the WOLFSSL object. This is useful when data needs to be read from a single place and demultiplexed into multiple connections. The caller should then call wolfSSL_read() to extract the plaintext data from the WOLFSSL object.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **data** data to inject into the ssl object.
- **sz** number of bytes of data to inject.

See: wolfSSL_read

Return:

- BAD_FUNC_ARG if any pointer parameter is NULL or sz <= 0
- BUFFER_ERROR if the input buffer lengths are inconsistent
- APP_DATA_READY if the input buffer must be grown while there is application data left to read
- MEMORY_E if allocation fails
- WOLFSSL_SUCCESS on success

Example

```
byte buf[2000]
sz = recv(fd, buf, sizeof(buf), 0);
if (sz <= 0)
    // error
if (wolfSSL_inject(ssl, buf, sz) != WOLFSSL_SUCCESS)
    // error
sz = wolfSSL_read(ssl, buf, sizeof(buf);
```

```
void * wolfSSL_GetIOReadCtx(
    WOLFSSL * ssl
)
```

This function returns the IOCB_ReadCtx member of the WOLFSSL struct.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_GetIOWriteCtx
- wolfSSL_SetIOReadFlags
- wolfSSL_SetIOWriteCtx
- wolfSSL_SetIOReadCtx
- wolfSSL_CTX_SetIOSend

Return:

- pointer This function returns a void pointer to the IOCB_ReadCtx member of the WOLFSSL structure.
- NULL returned if the WOLFSSL struct is NULL.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
void* ioRead;
...
ioRead = wolfSSL_GetIOReadCtx(ssl);
if(ioRead == NULL){
    // Failure case. The ssl object was NULL.
}
```

```
void * wolfSSL_GetIOWriteCtx(
    WOLFSSL * ssl
)
```

This function returns the IOCB_WriteCtx member of the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_GetIOReadCtx
- wolfSSL_SetIOWriteCtx
- wolfSSL_SetIOReadCtx
- wolfSSL_CTX_SetIOSend

Return:

- pointer This function returns a void pointer to the IOCB_WriteCtx member of the WOLFSSL structure.
- NULL returned if the WOLFSSL struct is NULL.

Example

```
WOLFSSL* ssl;
void* ioWrite;
```

```
...
ioWrite = wolfSSL_GetIOWriteCtx(ssl);
if(ioWrite == NULL){
    // The function returned NULL.
}
```

```
void wolfSSL_SetIO_NetX(
    WOLFSSL * ssl,
    NX_TCP_SOCKET * nxsocket,
    ULONG waitoption
)
```

This function sets the nxTcpSocket and nxWait members of the nxCtx struct within the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **nxsocket** a pointer to type NX_TCP_SOCKET that is set to the nxTcpSocket member of the nxCtx structure.
- **waitoption** a ULONG type that is set to the nxWait member of the nxCtx structure. With NX_NO_WAIT, or any wait option that can expire, the callbacks report WANT_READ/WANT_WRITE instead of a fatal error, so the wolfSSL call can be retried.

See:

- set_fd
- NetX_Send
- NetX_Receive

Return: none No returns.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
NX_TCP_SOCKET* nxsocket;
ULONG waitoption;
…
if(ssl != NULL || nxsocket != NULL || waitoption <= 0){
wolfSSL_SetIO_NetX(ssl, nxsocket, waitoption);
} else {
    // You need to pass in good parameters.
}
```

```
void wolfSSL_SetIO_NetX_Dtls(
    WOLFSSL * ssl,
    NX_UDP_SOCKET * nxsocket,
    NXD_ADDRESS nxdip,
    USHORT nxport,
```

```
    ULONG waitoption
)
```

This function configures the NetX Duo UDP context for a DTLS session. It stores the UDP socket, destination IP address (by value), destination port, and wait option into the WOLFSSL nxCtx structure. Requires WOLFSSL_NETX_DUO to be defined (ThreadX NetX Duo SDK).

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **nxsocket** a pointer to an NX_UDP_SOCKET already created and bound.
- **nxdip** the destination NXD_ADDRESS (passed by value; IPv4 or IPv6).
- **nxport** the destination UDP port number.
- **waitoption** a ULONG NetX wait option (e.g. NX_WAIT_FOREVER or ticks). With NX_NO_WAIT, or any wait option that can expire, NetX_SendTo reports WANT_WRITE instead of a fatal error. A receive that expires reports WANT_READ when the session is set non blocking, and a timeout otherwise so that DTLS can retransmit.

See:

- wolfSSL_SetIO_NetX
- NetX_SendTo
- NetX_ReceiveFrom

Return: none No returns.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
NX_UDP_SOCKET udpsocket;
NXD_ADDRESS   peeraddr;
USHORT        peerport = 4433;
ULONG         wait     = NX_WAIT_FOREVER;
// … initialise udpsocket and peeraddr …
wolfSSL_SetIO_NetX_Dtls(ssl, &udpsocket, peeraddr, peerport, wait);
```

```
int wolfIO_Select(
    SOCKET_T sockfd,
    int to_sec
)
```

Waits for socket to be ready for I/O with timeout.

Parameters:

- **sockfd** Socket file descriptor
- **to_sec** Timeout in seconds

See: wolfIO_TcpConnect

Return:

- 0 on success
- negative on error

Example

```
SOCKET_T sockfd;
int ret = wolfIO_Select(sockfd, 5);
```

```
int wolfIO_TcpConnect(
    SOCKET_T * sockfd,
    const char * ip,
    unsigned short port,
    int to_sec
)
```

Connects to TCP server with timeout.

Parameters:

- **sockfd** Pointer to socket file descriptor
- **ip** IP address string
- **port** Port number
- **to_sec** Timeout in seconds

See: wolfIO_TcpBind

Return:

- 0 on success
- negative on error

Example

```
SOCKET_T sockfd;
int ret = wolfIO_TcpConnect(&sockfd, "127.0.0.1", 443, 5);
```

```
int wolfIO_TcpAccept(
    SOCKET_T sockfd,
    SOCKADDR * peer_addr,
    XSOCKLENT * peer_len
)
```

Accepts TCP connection.

Parameters:

- **sockfd** Socket file descriptor
- **peer_addr** Peer address structure

- **peer_len** Peer address length

See: wolfIO_TcpBind

Return:

- Socket descriptor on success
- negative on error

Example

```
SOCKET_T sockfd;
SOCKADDR peer;
XSOCKLENT len = sizeof(peer);
int ret = wolfIO_TcpAccept(sockfd, &peer, &len);
```

```
int wolfIO_TcpBind(
    SOCKET_T * sockfd,
    word16 port
)
```

Binds TCP socket to port.

Parameters:

- **sockfd** Pointer to socket file descriptor
- **port** Port number

See: wolfIO_TcpAccept

Return:

- 0 on success
- negative on error

Example

```
SOCKET_T sockfd;
int ret = wolfIO_TcpBind(&sockfd, 443);
```

```
int wolfIO_Send(
    SOCKET_T sd,
    char * buf,
    int sz,
    int wrFlags
)
```

Sends data on socket.

Parameters:

- **sd** Socket descriptor
- **buf** Buffer to send
- **sz** Buffer size
- **wrFlags** Write flags

See: wolfIO_Recv

Return:

- Number of bytes sent on success
- negative on error

Example

```
SOCKET_T sd;
char buf[100];
int ret = wolfIO_Send(sd, buf, sizeof(buf), 0);
```

```
int wolfIO_Recv(
    SOCKET_T sd,
    char * buf,
    int sz,
    int rdFlags
)
```

Receives data from socket.

Parameters:

- **sd** Socket descriptor
- **buf** Buffer to receive into
- **sz** Buffer size
- **rdFlags** Read flags

See: wolfIO_Send

Return:

- Number of bytes received on success
- negative on error

Example

```
SOCKET_T sd;
char buf[100];
int ret = wolfIO_Recv(sd, buf, sizeof(buf), 0);
```

```
int wolfIO_SendTo(
    SOCKET_T sd,
    WOLFSSL_BIO_ADDR * addr,
    char * buf,
    int sz,
    int wrFlags
)
```

Sends datagram to address.

Parameters:

- **sd** Socket descriptor
- **addr** Destination address
- **buf** Buffer to send
- **sz** Buffer size
- **wrFlags** Write flags

See: wolfIO_RecvFrom

Return:

- Number of bytes sent on success
- negative on error

Example

```
SOCKET_T sd;
WOLFSSL_BIO_ADDR addr;
char buf[100];
int ret = wolfIO_SendTo(sd, &addr, buf, sizeof(buf), 0);
```

```
int wolfIO_RecvFrom(
    SOCKET_T sd,
    WOLFSSL_BIO_ADDR * addr,
    char * buf,
    int sz,
    int rdFlags
)
```

Receives datagram from address.

Parameters:

- **sd** Socket descriptor
- **addr** Source address
- **buf** Buffer to receive into
- **sz** Buffer size

- **rdFlags** Read flags

See: wolfIO_SendTo

Return:

- Number of bytes received on success
- negative on error

Example

```
SOCKET_T sd;
WOLFSSL_BIO_ADDR addr;
char buf[100];
int ret = wolfIO_RecvFrom(sd, &addr, buf, sizeof(buf), 0);
```

```
int wolfSSL_BioSend(
    WOLFSSL * ssl,
    char * buf,
    int sz,
    void * ctx
)
```

BIO send callback.

Parameters:

- **ssl** SSL object
- **buf** Buffer to send
- **sz** Buffer size
- **ctx** Context pointer

See: wolfSSL_BioReceive

Return:

- Number of bytes sent on success
- negative on error

Example

```
WOLFSSL* ssl;
char buf[100];
int ret = wolfSSL_BioSend(ssl, buf, sizeof(buf), NULL);
```

```
int wolfSSL_BioReceive(
    WOLFSSL * ssl,
    char * buf,
    int sz,
    void * ctx
)
```

BIO receive callback.

Parameters:

- **ssl** SSL object
- **buf** Buffer to receive into
- **sz** Buffer size
- **ctx** Context pointer

See: wolfSSL_BioSend

Return:

- Number of bytes received on success
- negative on error

Example

```
WOLFSSL* ssl;
char buf[100];
int ret = wolfSSL_BioReceive(ssl, buf, sizeof(buf), NULL);
```

```
int EmbedReceiveFromMcast(
    WOLFSSL * ssl,
    char * buf,
    int sz,
    void * ctx
)
```

Receives multicast datagram.

Parameters:

- **ssl** SSL object
- **buf** Buffer to receive into
- **sz** Buffer size
- **ctx** Context pointer

See: EmbedReceiveFrom

Return:

- Number of bytes received on success
- negative on error

Example

```
WOLFSSL* ssl;
char buf[100];
int ret = EmbedReceiveFromMcast(ssl, buf, sizeof(buf), NULL);
```

```
int wolfIO_HttpBuildRequestOcsp(
    const char * domainName,
    const char * path,
    int ocspReqSz,
    unsigned char * buf,
    int bufSize
)
```

Builds HTTP OCSP request.

Parameters:

- **domainName** Domain name
- **path** URL path
- **ocspReqSz** OCSP request size
- **buf** Output buffer
- **bufSize** Buffer size

See: wolfIO_HttpProcessResponseOcsp

Return:

- Request size on success
- negative on error

Example

```
char buf[1024];
int ret = wolfIO_HttpBuildRequestOcsp("example.com", "/ocsp", 100,
                                      (unsigned char*)buf, sizeof(buf));
```

```
int wolfIO_HttpProcessResponseOcspGenericIO(
    WolfSSLGenericIORecvCb ioCb,
    void * ioCbCtx,
    unsigned char ** respBuf,
    unsigned char * httpBuf,
    int httpBufSz,
    void * heap
)
```

Processes HTTP OCSP response with generic I/O.

Parameters:

- **ioCb** I/O callback
- **ioCbCtx** I/O callback context
- **respBuf** Response buffer pointer
- **httpBuf** HTTP buffer
- **httpBufSz** HTTP buffer size
- **heap** Heap hint

See: wolfIO_HttpProcessResponseOcsp

Return:

- 0 on success
- negative on error

Example

```
unsigned char* resp = NULL;
unsigned char httpBuf[1024];
int ret = wolfIO_HttpProcessResponseOcspGenericIO(myIoCb, ctx, &resp,
                                                  httpBuf,
                                                  sizeof(httpBuf), NULL);
```

```
int wolfIO_HttpProcessResponseOcsp(
    int sfd,
    unsigned char ** respBuf,
    unsigned char * httpBuf,
    int httpBufSz,
    void * heap
)
```

Processes HTTP OCSP response.

Parameters:

- **sfd** Socket file descriptor
- **respBuf** Response buffer pointer
- **httpBuf** HTTP buffer
- **httpBufSz** HTTP buffer size
- **heap** Heap hint

See: wolfIO_HttpBuildRequestOcsp

Return:

- 0 on success
- negative on error

Example

```
int sfd;
unsigned char* resp = NULL;
unsigned char httpBuf[1024];
int ret = wolfIO_HttpProcessResponseOcsp(sfd, &resp, httpBuf,
                                         sizeof(httpBuf), NULL);
```

```
int EmbedOcspLookup(
    void * ctx,
    const char * url,
    int urlSz,
    byte * ocspReqBuf,
    int ocspReqSz,
    byte ** ocspRespBuf
)
```

OCSP lookup callback.

Parameters:

- **ctx** Context pointer
- **url** URL string
- **urlSz** URL size
- **ocspReqBuf** OCSP request buffer
- **ocspReqSz** OCSP request size
- **ocspRespBuf** OCSP response buffer pointer

See: EmbedOcspRespFree

Return:

- 0 on success
- negative on error

Example

```
byte* resp = NULL;
byte req[100];
int ret = EmbedOcspLookup(NULL, "http://example.com/ocsp", 25, req,
                          sizeof(req), &resp);
```

```
int wolfIO_HttpBuildRequestCrl(
    const char * url,
    int urlSz,
    const char * domainName,
    unsigned char * buf,
    int bufSize
)
```

Builds HTTP CRL request.

Parameters:

- **url** URL string
- **urlSz** URL size
- **domainName** Domain name
- **buf** Output buffer
- **bufSize** Buffer size

See: wolfIO_HttpProcessResponseCrl

Return:

- Request size on success
- negative on error

Example

```
char buf[1024];
int ret = wolfIO_HttpBuildRequestCrl("http://example.com/crl", 22,
                                     "example.com",
                                     (unsigned char*)buf, sizeof(buf));
```

```
int wolfIO_HttpProcessResponseCrl(
    WOLFSSL_CRL * crl,
    int sfd,
    unsigned char * httpBuf,
    int httpBufSz
)
```

Processes HTTP CRL response.

Parameters:

- **crl** CRL object
- **sfd** Socket file descriptor
- **httpBuf** HTTP buffer
- **httpBufSz** HTTP buffer size

See: wolfIO_HttpBuildRequestCrl

Return:

- 0 on success
- negative on error

Example

```
WOLFSSL_CRL crl;
int sfd;
unsigned char httpBuf[1024];
```

```
int ret = wolfIO_HttpProcessResponseCrl(&crl, sfd, httpBuf,
                                        sizeof(httpBuf));
```

```
int EmbedCrlLookup(
    WOLFSSL_CRL * crl,
    const char * url,
    int urlSz
)
```

CRL lookup callback.

Parameters:

- **crl** CRL object
- **url** URL string
- **urlSz** URL size

See: wolfIO_HttpBuildRequestCrl

Return:

- 0 on success
- negative on error

Example

```
WOLFSSL_CRL crl;
int ret = EmbedCrlLookup(&crl, "http://example.com/crl", 22);
```

```
int wolfIO_DecodeUrl(
    const char * url,
    int urlSz,
    char * outName,
    char * outPath,
    unsigned short * outPort
)
```

Decodes URL into components.

Parameters:

- **url** URL string
- **urlSz** URL size
- **outName** Output domain name
- **outPath** Output path
- **outPort** Output port

See: wolfIO_HttpBuildRequest

Return:

- 0 on success
- negative on error

Example

```
char name[256], path[256];
unsigned short port;
int ret = wolfIO_DecodeUrl("http://example.com:443/path", 28, name,
                           path, &port);
```

```
int wolfIO_HttpBuildRequest(
    const char * reqType,
    const char * domainName,
    const char * path,
    int pathLen,
    int reqSz,
    const char * contentType,
    unsigned char * buf,
    int bufSize
)
```

Builds generic HTTP request.

Parameters:

- **reqType** Request type (GET, POST, etc.)
- **domainName** Domain name
- **path** URL path
- **pathLen** Path length
- **reqSz** Request body size
- **contentType** Content type
- **buf** Output buffer
- **bufSize** Buffer size

See: wolfIO_HttpProcessResponse

Return:

- Request size on success
- negative on error

Example

```
char buf[1024];
int ret = wolfIO_HttpBuildRequest("POST", "example.com", "/api", 4,
```

```
                                    100, "application/json",
                                    (unsigned char*)buf, sizeof(buf));
```

```
int wolfIO_HttpProcessResponseGenericIO(
    WolfSSLGenericIORecvCb ioCb,
    void * ioCbCtx,
    const char ** appStrList,
    unsigned char ** respBuf,
    unsigned char * httpBuf,
    int httpBufSz,
    int dynType,
    void * heap
)
```

Processes HTTP response with generic I/O.

Parameters:

- **ioCb** I/O callback
- **ioCbCtx** I/O callback context
- **appStrList** Application string list
- **respBuf** Response buffer pointer
- **httpBuf** HTTP buffer
- **httpBufSz** HTTP buffer size
- **dynType** Dynamic type
- **heap** Heap hint

See: wolfIO_HttpProcessResponse

Return:

- 0 on success
- negative on error

Example

```
unsigned char* resp = NULL;
unsigned char httpBuf[1024];
const char* appStrs[] = {"200 OK", NULL};
int ret = wolfIO_HttpProcessResponseGenericIO(myIoCb, ctx, appStrs,
                                              &resp, httpBuf,
                                              sizeof(httpBuf), 0, NULL);
```

```
int wolfIO_HttpProcessResponse(
    int sfd,
    const char ** appStrList,
    unsigned char ** respBuf,
    unsigned char * httpBuf,
    int httpBufSz,
```

```
    int dynType,
    void * heap
)
```

Processes HTTP response.

Parameters:

- **sfd** Socket file descriptor
- **appStrList** Application string list
- **respBuf** Response buffer pointer
- **httpBuf** HTTP buffer
- **httpBufSz** HTTP buffer size
- **dynType** Dynamic type
- **heap** Heap hint

See: wolfIO_HttpBuildRequest

Return:

- 0 on success
- negative on error

Example

```
int sfd;
unsigned char* resp = NULL;
unsigned char httpBuf[1024];
const char* appStrs[] = {"200 OK", NULL};
int ret = wolfIO_HttpProcessResponse(sfd, appStrs, &resp, httpBuf,
                                     sizeof(httpBuf), 0, NULL);
```

```
void wolfSSL_CTX_SetIOSend(
    WOLFSSL_CTX * ctx,
    CallbackIOSend CBIOSend
)
```

Sets I/O send callback for context.

Parameters:

- **ctx** SSL context
- **CBIOSend** Send callback

See: wolfSSL_SSLSetIOSend

Return: none No returns

Example

```
WOLFSSL_CTX* ctx;
wolfSSL_CTX_SetIOSend(ctx, mySendCallback);
```

```
void wolfSSL_SSLSetIORecv(
    WOLFSSL * ssl,
    CallbackIORecv CBIORecv
)
```

Sets I/O receive callback for SSL object.

Parameters:

- **ssl** SSL object
- **CBIORecv** Receive callback

See: wolfSSL_CTX_SetIORecv

Return: none No returns

Example

```
WOLFSSL* ssl;
wolfSSL_SSLSetIORecv(ssl, myRecvCallback);
```

```
void wolfSSL_SSLSetIOSend(
    WOLFSSL * ssl,
    CallbackIOSend CBIOSend
)
```

Sets I/O send callback for SSL object.

Parameters:

- **ssl** SSL object
- **CBIOSend** Send callback

See: wolfSSL_CTX_SetIOSend

Return: none No returns

Example

```
WOLFSSL* ssl;
wolfSSL_SSLSetIOSend(ssl, mySendCallback);
```

```
void wolfSSL_SetIO_Mynewt(
    WOLFSSL * ssl,
    struct mn_socket * mnSocket,
    struct mn_sockaddr_in * mnSockAddrIn
)
```

Sets I/O for Mynewt platform.

Parameters:

- **ssl** SSL object
- **mnSocket** Mynewt socket
- **mnSockAddrIn** Mynewt socket address

See: wolfSSL_SetIO_LwIP

Return: none No returns

Example

```
WOLFSSL* ssl;
struct mn_socket sock;
struct mn_sockaddr_in addr;
wolfSSL_SetIO_Mynewt(ssl, &sock, &addr);
```

```
int wolfSSL_SetIO_LwIP(
    WOLFSSL * ssl,
    void * pcb,
    tcp_recv_fn recv,
    tcp_sent_fn sent,
    void * arg
)
```

Sets I/O for LwIP platform.

Parameters:

- **ssl** SSL object
- **pcb** Protocol control block
- **recv** Receive callback
- **sent** Sent callback
- **arg** Argument pointer

See: wolfSSL_SetIO_Mynewt

Return:

- 0 on success
- negative on error

Example

```
WOLFSSL* ssl;
struct tcp_pcb* pcb;
int ret = wolfSSL_SetIO_LwIP(ssl, pcb, myRecv, mySent, NULL);
```

```
void wolfSSL_SetCookieCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

Sets cookie context for DTLS.

Parameters:

- **ssl** SSL object
- **ctx** Cookie context

See: wolfSSL_GetCookieCtx

Return: none No returns

Example

```
WOLFSSL* ssl;
void* ctx;
wolfSSL_SetCookieCtx(ssl, ctx);
```

```
void wolfSSL_CTX_SetIOGetPeer(
    WOLFSSL_CTX * ctx,
    CallbackGetPeer cb
)
```

Sets get peer callback for context.

Parameters:

- **ctx** SSL context
- **cb** Get peer callback

See: wolfSSL_CTX_SetIOSetPeer

Return: none No returns

Example

```
WOLFSSL_CTX* ctx;
wolfSSL_CTX_SetIOGetPeer(ctx, myGetPeerCallback);
```

```
void wolfSSL_CTX_SetIOSetPeer(
    WOLFSSL_CTX * ctx,
    CallbackSetPeer cb
)
```

Sets set peer callback for context.

Parameters:

- **ctx** SSL context
- **cb** Set peer callback

See: wolfSSL_CTX_SetIOGetPeer

Return: none No returns

Example

```
WOLFSSL_CTX* ctx;
wolfSSL_CTX_SetIOSetPeer(ctx, mySetPeerCallback);
```

```
int EmbedGetPeer(
    WOLFSSL * ssl,
    char * ip,
    int * ipSz,
    unsigned short * port,
    int * fam
)
```

Gets peer information.

Parameters:

- **ssl** SSL object
- **ip** IP address buffer
- **ipSz** IP address buffer size pointer
- **port** Port number pointer
- **fam** Address family pointer

See: EmbedSetPeer

Return:

- 0 on success
- negative on error

Example

```
WOLFSSL* ssl;
char ip[46];
int ipSz = sizeof(ip);
```

```
unsigned short port;
int fam;
int ret = EmbedGetPeer(ssl, ip, &ipSz, &port, &fam);
```

```
int EmbedSetPeer(
    WOLFSSL * ssl,
    char * ip,
    int ipSz,
    unsigned short port,
    int fam
)
```

Sets peer information.

Parameters:

- **ssl** SSL object
- **ip** IP address string
- **ipSz** IP address string size
- **port** Port number
- **fam** Address family

See: EmbedGetPeer

Return:

- 0 on success
- negative on error

Example

```
WOLFSSL* ssl;
int ret = EmbedSetPeer(ssl, "127.0.0.1", 9, 443, AF_INET);
```

A.6 wolfSSL Context and Session Set Up

A.5.2.114 function EmbedSetPeer

A.6.1 Types

	Name
typedef void()(*WOLFSSL* ssl, int client_server, int type, const byte *data, int len, void* arg)	**WOLFSSL_TLSEXT_DEBUG_CB** Callback type for the TLS extension debug callback.

A.6.2 Functions

	Name
WOLFSSL_METHOD *	**wolfSSLv23_method**(void)This function returns a WOLFSSL_METHOD similar to wolfSSLv23_client_method except that it is not determined which side yet (server/client).
WOLFSSL_METHOD *	**wolfSSLv3_server_method**(void)The wolfSSLv3_server_method() function is used to indicate that the application is a server and will only support the SSL 3.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().
WOLFSSL_METHOD *	**wolfSSLv3_client_method**(void)The wolfSSLv3_client_method() function is used to indicate that the application is a client and will only support the SSL 3.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().
WOLFSSL_METHOD *	**wolfTLSv1_server_method**(void)The wolfTLSv1_server_method() function is used to indicate that the application is a server and will only support the TLS 1.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().
WOLFSSL_METHOD *	**wolfTLSv1_client_method**(void)The wolfTLSv1_client_method() function is used to indicate that the application is a client and will only support the TLS 1.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().
WOLFSSL_METHOD *	**wolfTLSv1_1_server_method**(void)The wolfTLSv1_1_server_method() function is used to indicate that the application is a server and will only support the TLS 1.1 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().
WOLFSSL_METHOD *	**wolfTLSv1_1_client_method**(void)The wolfTLSv1_1_client_method() function is used to indicate that the application is a client and will only support the TLS 1.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

	Name
WOLFSSL_METHOD *	**wolfTLSv1_2_server_method**(void)The wolfTLSv1_2_server_method() function is used to indicate that the application is a server and will only support the TLS 1.2 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().
WOLFSSL_METHOD *	**wolfTLSv1_2_client_method**(void)The wolfTLSv1_2_client_method() function is used to indicate that the application is a client and will only support the TLS 1.2 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().
WOLFSSL_METHOD *	**wolfDTLSv1_client_method**(void)The wolfDTLSv1_client_method() function is used to indicate that the application is a client and will only support the DTLS 1.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). This function is only available when wolfSSL has been compiled with DTLS support (–enable_dtls, or by defining wolfSSL_DTLS).
WOLFSSL_METHOD *	**wolfDTLSv1_server_method**(void)The wolfDTLSv1_server_method() function is used to indicate that the application is a server and will only support the DTLS 1.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). This function is only available when wolfSSL has been compiled with DTLS support (–enable_dtls, or by defining wolfSSL_DTLS).
WOLFSSL_METHOD *	**wolfDTLSv1_3_server_method**(void)The wolfDTLSv1_3_server_method() function is used to indicate that the application is a server and will only support the DTLS 1.3 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). This function is only available when wolfSSL has been compiled with DTLSv1.3 support (–enable_dtls13, or by defining wolfSSL_DTLS13).

	Name
WOLFSSL_METHOD *	**wolfDTLSv1_3_client_method**(void)The wolfDTLSv1_3_client_method() function is used to indicate that the application is a client and will only support the DTLS 1.3 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). This function is only available when wolfSSL has been compiled with DTLSv1.3 support (–enable_dtls13, or by defining wolfSSL_DTLS13).
WOLFSSL_METHOD *	**wolfDTLS_server_method**(void)The wolfDTLS_server_method() function is used to indicate that the application is a server and will support the highest version of DTLS available and all the version up to the minimum version allowed. The default minimum version allowed is based on the define WOLFSSL_MIN_DTLS_DOWNGRADE and can be changed at runtime using wolfSSL_SetMinVersion(). This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). This function is only available when wolfSSL has been compiled with DTLS support (–enable_dtls, or by defining wolfSSL_DTLS).
WOLFSSL_METHOD *	**wolfDTLS_client_method**(void)The wolfDTLS_client_method() function is used to indicate that the application is a client and will support the highest version of DTLS available and all the version up to the minimum version allowed. The default minimum version allowed is based on the define WOLFSSL_MIN_DTLS_DOWNGRADE and can be changed at runtime using wolfSSL_SetMinVersion(). This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). This function is only available when wolfSSL has been compiled with DTLS support (–enable_dtls, or by defining wolfSSL_DTLS).
int	**wolfSSL_use_old_poly**(WOLFSSL * ssl, int value)Since there is some differences between the first release and newer versions of chacha-poly AEAD construction we have added an option to communicate with servers/clients using the older version. By default wolfSSL uses the new version.

	Name
int	**wolfSSL_CTX_trust_peer_cert**(WOLFSSL_CTX * ctx, const char * file, int type)This function loads a certificate to use for verifying a peer when performing a TLS/SSL handshake. The peer certificate sent during the handshake is compared by using the SKID when available and the signature. If these two things do not match then any loaded CAs are used. Feature is enabled by defining the macro WOLFSSL_TRUST_PEER_CERT. Please see the examples for proper usage.
long	**wolfSSL_CTX_get_verify_depth**(WOLFSSL_CTX * ctx)This function gets the certificate chaining depth using the CTX structure.
WOLFSSL_CTX *	**wolfSSL_CTX_new**(WOLFSSL_METHOD *)This function creates a new SSL context, taking a desired SSL/TLS protocol method for input.
WOLFSSL *	**wolfSSL_new**(WOLFSSL_CTX *)This function creates a new SSL session, taking an already created SSL context as input.
int	**wolfSSL_set_fd**(WOLFSSL * ssl, int fd)This function assigns a file descriptor (fd) as the input/output facility for the SSL connection. Typically this will be a socket file descriptor.
int	**wolfSSL_set_dtls_fd_connected**(WOLFSSL * ssl, int fd)This function assigns a file descriptor (fd) as the input/output facility for the SSL connection. Typically this will be a socket file descriptor. This is a DTLS specific API because it marks that the socket is connected. recvfrom and sendto calls on this fd will have the addr and addr_len parameters set to NULL.

	Name
int	**wolfDTLS_SetChGoodCb**(WOLFSSL * ssl, ClientHelloGoodCb cb, void * user_ctx)Allows setting a callback for a correctly processed and verified DTLS client hello. When using a cookie exchange mechanism (either the HelloVerifyRequest in DTLS 1.2 or the HelloRetryRequest with a cookie extension in DTLS 1.3) this callback is called after the cookie exchange has succeeded. This is useful to use one WOLFSSL object as the listener for new connections and being able to isolate the WOLFSSL object once the ClientHello is verified (either through a cookie exchange or just checking if the ClientHello had the correct format). With DTLS cookies disabled, the callback is invoked once per ClientHello, only after complete successful ClientHello processing, not on its first fragment. Retries, retransmitted fragments and a second ClientHello after a key_share HelloRetryRequest do not repeat the notification. The object is already stateful before input is read, but the peer's return_routability has not been verified. Returning 0 accepts the ClientHello and the handshake continues. Any negative return stops the accept call with that error and leaves the accept state machine at the ClientHello, so nothing is sent to the peer and the next accept call asks the callback again instead of continuing the handshake. All negative returns behave this way; the value only decides what wolfSSL_get_error() reports, so what it means is the application's choice. A callback that is not ready to decide can return WANT_READ or WANT_WRITE and be asked again as many times as it needs, and one that refuses the peer returns its own error and keeps the server silent by refusing again on every later accept call. The cookie mode cannot be changed from the callback; see wolfSSL_disable_cookie(). DTLS 1.2: https://datatracker.ietf.org/doc/html/rfc6347#section_4.2.1 DTLS 1.3: https://www.rfc_editor.org/rfc/rfc8446#section_4.2.2.

	Name
void	**wolfSSL_set_using_nonblock**(WOLFSSL * ssl, int nonblock)This function informs the WOLFSSL object that the underlying I/O is non_blocking. After an application creates a WOLFSSL object, if it will be used with a non_blocking socket, call wolfSSL_set_using_nonblock() on it. This lets the WOLFSSL object know that receiving EWOULDBLOCK means that the recvfrom call would block rather than that it timed out.
int	**wolfSSL_disable_cookie**(WOLFSSL * ssl)Disable server cookies for DTLS 1.2, DTLS 1.3 and TLS 1.3, including DTLS 1.2 fallback from DTLS 1.3. Cookies are enabled by default for DTLS. For DTLS, ordinary wolfSSL_accept() or wolfSSL_accept_TLSv13() commits it to stateful processing before the first read, even if a nonblocking accept has no input available. The application must isolate/demultiplex the peer before accepting; disabling cookies removes return_routability verification and exposes the server to DoS/amplification attacks. wolfDTLS_accept_stateless() cannot be used.
int	**wolfSSL_enable_cookie**(WOLFSSL * ssl)Enable server cookies for DTLS 1.2, DTLS 1.3 and TLS 1.3. Like wolfSSL_disable_cookie(), this does not change stateful processing. It can undo a disable before accept begins, including wolfSSL_disable_hrr_cookie(). Missing primary cookie secrets are randomly generated immediately; existing primary and secondary secrets are preserved.
void	**wolfSSL_CTX_free**(WOLFSSL_CTX * ctx)This function frees an allocated WOLFSSL_CTX object. This function decrements the CTX reference count and only frees the context when the reference count has reached 0.
void	**wolfSSL_free**(WOLFSSL * ssl)This function frees an allocated wolfSSL object.

	Name
int	**wolfSSL_set_session**(WOLFSSL * ssl, WOLFSSL_SESSION * session)This function sets the session to be used when the SSL object, ssl, is used to establish a SSL/TLS connection. For session resumption, before calling wolfSSL_shutdown() with your session object, an application should save the session ID from the object with a call to wolfSSL_get1_session(), which returns a pointer to the session. Later, the application should create a new WOLFSSL object and assign the saved session with wolfSSL_set_session(). At this point, the application may call wolfSSL_connect() and wolfSSL will try to resume the session. The wolfSSL server code allows session resumption by default. The object returned by wolfSSL_get1_session() needs to be freed after the application is done with it by calling wolfSSL_SESSION_free() on it.

	Name
void	**wolfSSL_CTX_set_verify**(WOLFSSL_CTX * ctx, int mode, VerifyCallback verify_callback)This function sets the verification method for remote peers and also allows a verify callback to be registered with the SSL context. The verify callback will be called only when a verification failure has occurred. If no verify callback is desired, the NULL pointer can be used for verify_callback. The verification mode of peer certificates is a logically OR'd list of flags. The possible flag values include: SSL_VERIFY_NONE Client mode: the client will not verify the certificate received from the server and the handshake will continue as normal. Server mode: the server will not send a certificate request to the client. As such, client verification will not be enabled. SSL_VERIFY_PEER Client mode: the client will verify the certificate received from the server during the handshake. This is turned on by default in wolfSSL, therefore, using this option has no effect. Server mode: the server will send a certificate request to the client and verify the client certificate received. SSL_VERIFY_FAIL_IF_NO_PEER_CERT Client mode: no effect when used on the client side. Server mode: the verification will fail on the server side if the client fails to send a certificate when requested to do so (when using SSL_VERIFY_PEER on the SSL server). SSL_VERIFY_FAIL_EXCEPT_PSK Client mode: no effect when used on the client side. Server mode: the verification is the same as SSL_VERIFY_FAIL_IF_NO_PEER_CERT except in the case of a PSK connection. If a PSK connection is being made then the connection will go through without a peer cert. SSL_VERIFY_CLIENT_ONCE Accepted and ignored. It is present so that OpenSSL_derived code compiles unchanged; wolfSSL does not act on it, does not store it, and wolfSSL_get_verify_mode() / wolfSSL_CTX_get_verify_mode() will not report it back. Including it in mode has no effect on any of the other flags.

	Name
void	**wolfSSL_set_verify**(WOLFSSL * ssl, int mode, VerifyCallback verify_callback)This function sets the verification method for remote peers and also allows a verify callback to be registered with the SSL session. The verify callback will be called only when a verification failure has occurred. If no verify callback is desired, the NULL pointer can be used for verify_callback. The verification mode of peer certificates is a logically OR'd list of flags. The possible flag values include: SSL_VERIFY_NONE Client mode: the client will not verify the certificate received from the server and the handshake will continue as normal. Server mode: the server will not send a certificate request to the client. As such, client verification will not be enabled. SSL_VERIFY_PEER Client mode: the client will verify the certificate received from the server during the handshake. This is turned on by default in wolfSSL, therefore, using this option has no effect. Server mode: the server will send a certificate request to the client and verify the client certificate received. SSL_VERIFY_FAIL_IF_NO_PEER_CERT Client mode: no effect when used on the client side. Server mode: the verification will fail on the server side if the client fails to send a certificate when requested to do so (when using SSL_VERIFY_PEER on the SSL server). SSL_VERIFY_FAIL_EXCEPT_PSK Client mode: no effect when used on the client side. Server mode: the verification is the same as SSL_VERIFY_FAIL_IF_NO_PEER_CERT except in the case of a PSK connection. If a PSK connection is being made then the connection will go through without a peer cert. SSL_VERIFY_CLIENT_ONCE Accepted and ignored. It is present so that OpenSSL_derived code compiles unchanged; wolfSSL does not act on it, does not store it, and wolfSSL_get_verify_mode() / wolfSSL_CTX_get_verify_mode() will not report it back. Including it in mode has no effect on any of the other flags.

	Name
long	**wolfSSL_CTX_set_session_cache_mode**(WOLFSSL_CTX * ctx, long mode)This function enables or disables SSL session caching. Behavior depends on the value used for mode. The following values for mode are available: SSL_SESS_CACHE_OFF- disable session caching. Session caching is turned on by default. SSL_SESS_CACHE_NO_AUTO_CLEAR - Disable auto-flushing of the session cache. Auto-flushing is turned on by default.
void	**wolfSSL_CTX_sess_set_get_cb**(WOLFSSL_CTX * ctx, WOLFSSL_SESSION *()*(WOLFSSL *ssl, const unsigned char* , int, int *) f)This function registers the callback wolfSSL uses to look a session up in an application managed (external) session cache. It is called on the server side when a peer offers a session id to resume, before the internal cache is consulted. Compatibility layer equivalent of SSL_CTX_sess_set_get_cb().
int	**wolfSSL_CTX_memrestore_cert_cache**(WOLFSSL_CTX * ctx, const void * mem, int sz)This function restores the certificate cache from memory.
int	**wolfSSL_CTX_set_cipher_list**(WOLFSSL_CTX * ctx, const char * list)This function sets the cipher suite list for a given WOLFSSL_CTX. The list becomes the default cipher suite list for any new WOLFSSL sessions created from the context, and the order in the string is the local preference order from highest to lowest. Each call replaces the previous list. The list is a null-terminated, colon-delimited text string of suite names and/or OpenSSL-style group keywords, for example “TLS13-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:DHE-RSA-AES256-SHA256”.
int	**wolfSSL_set_cipher_list**(WOLFSSL * ssl, const char * list)This function sets the cipher suite list for a given WOLFSSL session. The list format and the set of recognized suite names and keywords are identical to those documented for wolfSSL_CTX_set_cipher_list(); refer to that function for the full keyword/suite tables and required build options. Each call replaces the previous list on the session.

	Name
void	**wolfSSL_dtls13_set_send_more_acks**(WOLFSSL * ssl, int value)This function sets whether the library should send ACKs to the other peer immediately when detecting disruption or not. Sending ACKs immediately assures minimum latency but it may consume more bandwidth than necessary. If the application manages the timer by itself and this option is set to 0 then application code can use wolfSSL_dtls13_use_quick_timeout() to determine if it should setup a quicker timeout to send those delayed ACKs.
int	**wolfSSL_dtls_set_timeout_init**(WOLFSSL * ssl, int timeout)This function sets the dtls timeout.
int	**wolfSSL_dtls13_do_scheduled_work**(WOLFSSL * ssl)Sends the DTLS 1.3 work that was scheduled while reading. With WOLFSSL_RW_THREADED the read path never transmits, because that would race the write thread over the output buffer and the sending key schedule, so ACKs, retransmissions and a KeyUpdate the peer asked for are only sent from the write side. An application that reads without writing must call this, from the same thread it uses for writing, or those messages are never sent and the peer keeps retransmitting what it is waiting to have acknowledged. Not for write_dup applications, which drain through wolfSSL_write().
int	**wolfSSL_dtls13_pending_work**(WOLFSSL * ssl)Reports whether the object has DTLS 1.3 work waiting to be sent by wolfSSL_dtls13_do_scheduled_work(). Only meaningful with WOLFSSL_RW_THREADED. The answer is advisory and can change as soon as it is returned. Only work the pump can actually carry out is reported, so a drain loop over the pair terminates; waiting for the peer to acknowledge a key update we sent is not reported, as there is nothing to send for it. A record the pump could only write in part is reported, so that the retry it owes is not lost.

	Name
WOLFSSL_SESSION *	**wolfSSL_get1_session**(WOLFSSL * ssl)This function returns the WOLFSSL_SESSION from the WOLFSSL structure as a reference type. This requires calling wolfSSL_SESSION_free to release the session reference. The WOLFSSL_SESSION pointed to contains all the necessary information required to perform a session resumption and reestablish the connection without a new handshake. For session resumption, before calling wolfSSL_shutdown() with your session object, an application should save the session ID from the object with a call to wolfSSL_get1_session(), which returns a pointer to the session. Later, the application should create a new WOLFSSL object and assign the saved session with wolfSSL_set_session(). At this point, the application may call wolfSSL_connect() and wolfSSL will try to resume the session. The wolfSSL server code allows session resumption by default. The object returned by wolfSSL_get1_session() needs to be freed after the application is done with it by calling wolfSSL_SESSION_free() on it.
WOLFSSL_METHOD *	**wolfSSLv23_client_method**(void)The wolfSSLv23_client_method() function is used to indicate that the application is a client and will support the highest protocol version supported by the server between SSL 3.0 _ TLS 1.3. This function allocates memory for and initializes a new WOLFSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). Both wolfSSL clients and servers have robust version downgrade capability. If a specific protocol version method is used on either side, then only that version will be negotiated or an error will be returned. For example, a client that uses TLSv1 and tries to connect to a SSLv3 only server will fail, likewise connecting to a TLSv1.1 will fail as well. To resolve this issue, a client that uses the wolfSSLv23_client_method() function will use the highest protocol version supported by the server and downgrade to SSLv3 if needed. In this case, the client will be able to connect to a server running SSLv3 - TLSv1.3.
WOLFSSL_BIGNUM *	**wolfSSL_ASN1_INTEGER_to_BN**(const WOLFSSL_ASN1_INTEGER * ai, WOLFSSL_BIGNUM * bn)This function is used to copy a WOLFSSL_ASN1_INTEGER value to a WOLFSSL_BIGNUM structure.

	Name
long	**wolfSSL_CTX_add_extra_chain_cert**(WOLFSSL_CTX * ctx, WOLFSSL_X509 * x509)This function adds the certificate to the internal chain being built in the WOLFSSL_CTX structure.
int	**wolfSSL_CTX_get_read_ahead**(WOLFSSL_CTX * ctx)This function returns the get read ahead flag from a WOLFSSL_CTX structure.
int	**wolfSSL_CTX_set_read_ahead**(WOLFSSL_CTX * ctx, int v)This function sets the read ahead flag in the WOLFSSL_CTX structure. When enabled, the record_header read pulls in up to a full TLS record in a single recv(), so the body (and any following buffered records) is obtained without a second syscall. The flag only changes I/O behaviour when the library is built with read_ahead support (–enable_readahead / WOLFSSL_TLS_READ_AHEAD); otherwise it is stored but inert.
int	**wolfSSL_CTX_set_default_read_buffer_len**(WOLFSSL_CTX * ctx, size_t len)This function sets the read_ahead receive window size for contexts created from this WOLFSSL_CTX. When read_ahead is enabled (wolfSSL_CTX_set_read_ahead()), a single recv() pulls in up to len bytes:
int	**wolfSSL_set_default_read_buffer_len**(WOLFSSL * ssl, size_t len)This function sets the read_ahead coalescing buffer size on a single WOLFSSL session, overriding the value inherited from its WOLFSSL_CTX. See wolfSSL_CTX_set_default_read_buffer_len() for the full description, the one-record default, and the memory-vs-syscall trade-off.
long	**wolfSSL_CTX_get_default_read_buffer_len**(WOLFSSL_CTX * ctx)This function returns the read_ahead coalescing buffer size configured on a WOLFSSL_CTX by wolfSSL_CTX_set_default_read_buffer_len(). Because a length of 0 and oversized values are normalised when set, the value reported is the effective window actually in use (the one_record default rather than 0 when the length was never changed), not the raw argument last passed.

	Name
long	**wolfSSL_get_default_read_buffer_len**(const WOLFSSL * ssl)This function returns the read_ahead coalescing buffer size in effect for a single WOLFSSL session, whether inherited from its WOLFSSL_CTX or overridden by wolfSSL_set_default_read_buffer_len(). As with the WOLFSSL_CTX getter, the value reported is the effective window in use, not the raw argument last passed.
int	**wolfSSL_CTX_set_max_send_fragment**(WOLFSSL_CTX * ctx, long max_fragment)This function caps the plaintext carried in each record sent by WOLFSSL objects created from the context afterwards. It is the wolfSSL equivalent of OpenSSL's SSL_CTX_set_max_send_fragment(). The cap is local and send only: the peer is not told and records received are not limited. Unset, there is no cap. wolfSSL_new() copies the value, so a later call does not change existing objects, as in OpenSSL. With a negotiated max_fragment_length the smaller of the two limits applies, where OpenSSL uses the negotiated length instead. A DTLS write larger than the cap fails with DTLS_SIZE_ERROR, like one larger than the MTU, unless WOLFSSL_NO_DTLS_SIZE_CHECK is defined. In TLS 1.3 only Certificate and CertificateVerify are split to the cap. Other handshake messages, such as ClientHello, ServerHello, EncryptedExtensions, CertificateRequest and NewSessionTicket, are sent in one record each, as they are with a negotiated max_fragment_length. Application data is always split.
int	**wolfSSL_set_max_send_fragment**(WOLFSSL * ssl, long max_fragment)This function caps the plaintext carried in each record sent by a single WOLFSSL object, replacing the value copied from its WOLFSSL_CTX. It is the wolfSSL equivalent of OpenSSL's SSL_set_max_send_fragment() and takes effect from the next record sent. See wolfSSL_CTX_set_max_send_fragment() for the full description.
long	**wolfSSL_CTX_set_tlsext_opaque_prf_input_callback_arg**(WOLF * ctx, void * arg)This function sets the optional argument to be passed to the PRF callback.

	Name
long	**wolfSSL_CTX_set_options**(WOLFSSL_CTX * ctx, long opt)This function ORs the bits in opt into the options mask of the given WOLFSSL_CTX. The options mask is inherited by every WOLFSSL session later created from this context. Bits are accumulated — to remove an option, use wolfSSL_CTX_clear_options(). The OpenSSL-style "SSL_OP_" *macros are aliases for the corresponding "WOLFSSL_OP_"* values; either spelling may be used.
long	**wolfSSL_set_options**(WOLFSSL * s, long op)This function ORs the bits in op into the options mask of the given WOLFSSL session. The set of recognized "SSL_OP_" / *"WOLFSSL_OP_"* flags is identical to that documented for wolfSSL_CTX_set_options(); refer to that function for the full options table and the build-option requirements of individual flags. Flags inherited from the parent WOLFSSL_CTX remain set; setting SSL_OP_NO_TLSv1_3 here lowers the session's negotiated minor version to TLS 1.2.
long	**wolfSSL_get_options**(const WOLFSSL * s)This function returns the current options mask.
long	**wolfSSL_set_tlsext_debug_arg**(WOLFSSL * s, void * arg)This is used to set the debug argument passed around.
long	**wolfSSL_set_tlsext_debug_callback**(WOLFSSL * s, WOLFSSL_TLSEXT_DEBUG_CB cb)This is used to set the TLS extension debug callback on the object.
long	**wolfSSL_get_verify_result**(const WOLFSSL * ssl)This is used to get the results after trying to verify the peer's certificate.
int	**wolfSSL_CTX_allow_anon_cipher**(WOLFSSL_CTX * ctx)This function enables the havAnon member of the CTX structure if HAVE_ANON is defined during compilation.
WOLFSSL_METHOD *	**wolfSSLv23_server_method**(void)The wolfSSLv23_server_method() function is used to indicate that the application is a server and will support clients connecting with protocol version from SSL 3.0 _ TLS 1.3. This function allocates memory for and initializes a new WOLFSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().
int	**wolfSSL_state**(WOLFSSL * ssl)This is used to get the internal error state of the WOLFSSL structure.

	Name
int	**wolfSSL_check_domain_name**(WOLFSSL * ssl, const char * dn)wolfSSL by default checks the peer certificate for a valid date range and a verified signature. Calling this function before wolfSSL_connect() or wolfSSL_accept() will add a domain name check to the list of checks to perform. dn holds the domain name to check against the peer certificate when it's received.
int	**wolfSSL_check_ip_address**(WOLFSSL * ssl, const char * ipaddr)Calling this function before wolfSSL_connect() or wolfSSL_accept() adds an IP-address identity check against the peer certificate SAN iPAddress entries.
int	**wolfSSL_set_compression**(WOLFSSL * ssl)Turns on the ability to use compression for the SSL connection. Both sides must have compression turned on otherwise compression will not be used. The zlib library performs the actual data compression. To compile into the library use –with-libz for the configure system and define HAVE_LIBZ otherwise. Keep in mind that while compressing data before sending decreases the actual size of the messages being sent and received, the amount of data saved by compression usually takes longer in time to analyze than it does to send it raw on all but the slowest of networks.
int	**wolfSSL_set_timeout**(WOLFSSL * ssl, unsigned int to)This function sets the SSL session timeout value in seconds.
int	**wolfSSL_CTX_set_timeout**(WOLFSSL_CTX * ctx, unsigned int to)This function sets the timeout value for SSL sessions, in seconds, for the specified SSL context.
int	**wolfSSL_CTX_UnloadCAs**(WOLFSSL_CTX * ctx)This function unloads the CA signer list and frees the whole signer table.
int	**wolfSSL_CTX_UnloadIntermediateCerts**(WOLFSSL_CTX * ctx)This function unloads intermediate certificates added to the CA signer list and frees them.
int	**wolfSSL_CTX_Unload_trust_peers**(WOLFSSL_CTX * ctx)This function is used to unload all previously loaded trusted peer certificates. Feature is enabled by defining the macro WOLFSSL_TRUST_PEER_CERT.

	Name
int	**wolfSSL_CTX_trust_peer_buffer**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format)This function loads a certificate to use for verifying a peer when performing a TLS/SSL handshake. The peer certificate sent during the handshake is compared by using the SKID when available and the signature. If these two things do not match then any loaded CAs are used. Is the same functionality as wolfSSL_CTX_trust_peer_cert except is from a buffer instead of a file. Feature is enabled by defining the macro WOLFSSL_TRUST_PEER_CERT Please see the examples for proper usage.
int	**wolfSSL_CTX_set_group_messages**(WOLFSSL_CTX * ctx)This function turns on grouping of handshake messages where possible.
int	**wolfSSL_set_group_messages**(WOLFSSL * ssl)This function turns on grouping of handshake messages where possible.
int	**wolfSSL_CTX_SetMinVersion**(WOLFSSL_CTX * ctx, int version)This function sets the minimum downgrade version allowed. Applicable only when the connection allows downgrade using (wolfSSLv23_client_method or wolfSSLv23_server_method).
int	**wolfSSL_SetVersion**(WOLFSSL * ssl, int version)This function sets the SSL/TLS protocol version for the specified SSL session (WOLFSSL object) using the version as specified by version. This will override the protocol setting for the SSL session (ssl) _ originally defined and set by the SSL context (wolfSSL_CTX_new()) method type.
int	**wolfSSL_UseALPN**(WOLFSSL * ssl, char * protocol_name_list, unsigned int protocol_name_listSz, unsigned char options)Setup ALPN use for a wolfSSL session.
int	**wolfSSL_CTX_UseSessionTicket**(WOLFSSL_CTX * ctx)This function sets wolfSSL context to use a session ticket.
int	**wolfSSL_check_private_key**(const WOLFSSL * ssl)This function checks that the private key is a match with the certificate being used.
int	**wolfSSL_use_certificate**(WOLFSSL * ssl, WOLFSSL_X509 * x509)his is used to set the certificate for WOLFSSL structure to use during a handshake.

	Name
int	**wolfSSL_use_certificate_ASN1**(WOLFSSL * ssl, const unsigned char * der, int derSz)This is used to set the certificate for WOLFSSL structure to use during a handshake. A DER formatted buffer is expected.
int	**wolfSSL_SESSION_get_master_key**(const WOLFSSL_SESSION * ses, unsigned char * out, int outSz)This is used to get the master key after completing a handshake.
int	**wolfSSL_SESSION_get_master_key_length**(const WOLFSSL_SESSION * ses)This is used to get the master secret key length.
void	**wolfSSL_CTX_set_cert_store**(WOLFSSL_CTX * ctx, WOLFSSL_X509_STORE * str)This is a setter function for the WOLFSSL_X509_STORE structure in ctx.
WOLFSSL_X509_STORE *	**wolfSSL_CTX_get_cert_store**(WOLFSSL_CTX * ctx)This is a getter function for the WOLFSSL_X509_STORE structure in ctx.
size_t	**wolfSSL_get_server_random**(const WOLFSSL * ssl, unsigned char * out, size_t outlen)This is used to get the random data sent by the server during the handshake.
size_t	**wolfSSL_get_client_random**(const WOLFSSL * ssl, unsigned char * out, size_t outSz)This is used to get the random data sent by the client during the handshake.
wc_pem_password_cb *	**wolfSSL_CTX_get_default_passwd_cb**(WOLFSSL_CTX * ctx)This is a getter function for the password callback set in ctx.
void *	**wolfSSL_CTX_get_default_passwd_cb_userdata**(WOLFSSL_CTX * ctx)This is a getter function for the password callback user data set in ctx.
long	**wolfSSL_CTX_clear_options**(WOLFSSL_CTX * ctx, long opt)This function resets option bits of WOLFSSL_CTX object.
int	**wolfSSL_set_msg_callback**(WOLFSSL * ssl, SSL_Msg_Cb cb)This function sets a callback in the ssl. The callback is to observe handshake messages. NULL value of cb resets the callback.
int	**wolfSSL_set_msg_callback_arg**(WOLFSSL * ssl, void * arg)This function sets associated callback context value in the ssl. The value is handed over to the callback argument.

	Name
int	**wolfSSL_send_hrr_cookie**(WOLFSSL * ssl, const unsigned char * secret, unsigned int secretSz)This function is called on the server side to indicate that a HelloRetryRequest message must contain a Cookie and, in case of using protocol DTLS v1.3, that the handshake will always include a cookie exchange. Please note that when using protocol DTLS v1.3, the cookie exchange is enabled by default. The Cookie holds a hash of the current transcript so that another server process can handle the ClientHello in reply. The secret is used when generating the integrity check on the Cookie data. This replaces or regenerates the HRR secret, then delegates to wolfSSL_enable_cookie(). Changing cookie mode after handshake processing starts is unsupported; rotating a secret while cookies are already enabled remains supported. DTLS 1.2_only callers should use wolfSSL_enable_cookie() and wolfSSL_DTLS_SetCookieSecret().
int	**wolfSSL_set_hrr_cookie_secret_secondary**(WOLFSSL * ssl, const unsigned char * secret, unsigned int secretSz)This function sets a secondary HelloRetryRequest cookie secret on a DTLS 1.3 server. It is used only when verifying a received cookie, and only if the primary secret (set with wolfSSL_send_hrr_cookie()) fails to verify it. This lets an application rotate the cookie secret on a stateless DTLS 1.3 server without rejecting clients whose cookie was issued under the previous secret: install the new secret as the primary and the previous secret here for an overlap window. The secondary secret is never used to issue cookies. This API is DTLS only; TLS 1.3 over a reliable transport does not verify cookies statelessly across the HelloRetryRequest exchange.

	Name
int	**wolfSSL_disable_hrr_cookie**(WOLFSSL * ssl)This function is called on the server side to indicate that a HelloRetryRequest message must NOT contain a Cookie and that, if using protocol DTLS v1.3, a cookie exchange will not be included in the handshake. Please note that not doing a cookie exchange when using protocol DTLS v1.3 can make the server susceptible to DoS/Amplification attacks. This delegates to wolfSSL_disable_cookie(): on a DTLS object it also disables the DTLS 1.2 HelloVerifyRequest exchange, and it fails with BAD_STATE_E once the handshake has committed to stateful processing.
int	**wolfSSL_CTX_no_ticket_TLSv13**(WOLFSSL_CTX * ctx)This function is called on the server to stop it from sending a resumption session ticket once the handshake is complete.
int	**wolfSSL_no_ticket_TLSv13**(WOLFSSL * ssl)This function is called on the server to stop it from sending a resumption session ticket once the handshake is complete.
int	**wolfSSL_CTX_no_dhe_psk**(WOLFSSL_CTX * ctx)This function is called on a TLS v1.3 wolfSSL context to disallow Diffie_Hellman (DH) style key exchanges when handshakes are using pre-shared keys for authentication.
int	**wolfSSL_no_dhe_psk**(WOLFSSL * ssl)This function is called on a TLS v1.3 client or server wolfSSL to disallow Diffie_Hellman (DH) style key exchanges when handshakes are using pre-shared keys for authentication.

	Name
int	**wolfSSL_CTX_require_psk**(WOLFSSL_CTX * ctx)This function is called on a TLS v1.3 / DTLS v1.3 context to require that an external Pre_Shared Key is negotiated for the handshake to succeed. When set, a handshake that completes without negotiating an external PSK is aborted with PSK_MISSING_ERROR instead of falling back to a certificate handshake, so the PSK acts as an additional security factor. The requirement keys off the external_PSK callback (it has no effect unless one is registered) and session_ticket resumption is exempt. To preserve forward secrecy a mandatory external PSK must also use an (EC)DHE key exchange; a pure psk_ke handshake is rejected with PSK_KEY_ERROR. This applies to TLS 1.3 and DTLS 1.3 only; in (D)TLS 1.2 the use of a PSK is determined by the negotiated cipher suite, so a mandatory PSK is instead configured by restricting the cipher suite list to (preferably (EC)DHE_)PSK suites.
int	**wolfSSL_require_psk**(WOLFSSL * ssl)This function is called on a TLS v1.3 / DTLS v1.3 wolfSSL object to require that an external Pre_Shared Key is negotiated for the handshake to succeed. See wolfSSL_CTX_require_psk() for the full behaviour.
int	**wolfSSL_CTX_allow_post_handshake_auth**(WOLFSSL_CTX * ctx)This function is called on a TLS v1.3 client wolfSSL context to allow a client certificate to be sent post handshake upon request from server. This is useful when connecting to a web server that has some pages that require client authentication and others that don't.
int	**wolfSSL_allow_post_handshake_auth**(WOLFSSL * ssl)This function is called on a TLS v1.3 client wolfSSL to allow a client certificate to be sent post handshake upon request from server. A Post-Handshake Client Authentication extension is sent in the ClientHello. This is useful when connecting to a web server that has some pages that require client authentication and others that don't.

	Name
int	**wolfSSL_CTX_set1_sigalgs_list**(WOLFSSL_CTX * ctx, const char * list)This function sets the list of signature algorithms to allow on a wolfSSL context in order of preference. The list is a null_terminated text string and a colon_delimited list, where each element is of the form “+” (for example “RSA_PSS+SHA256:ECDSA+SHA384”). The Edwards-curve algorithms “ED25519” and “ED448” are written without a digest suffix, since the hash is implied by the algorithm. The previous list stored in the context is replaced.
int	**wolfSSL_set1_sigalgs_list**(WOLFSSL * ssl, const char * list)This function sets the list of signature algorithms to allow on a wolfSSL session in order of preference. The list format and the set of recognized public_key and digest tokens are identical to those documented for wolfSSL_CTX_set1_sigalgs_list(); refer to that function for the full token tables and TLS 1.3 caveats. The previous list stored in the session is replaced.
int	**wolfSSL_CTX_set1_groups_list**(WOLFSSL_CTX * ctx, const char * list)This function sets the list of key_exchange groups (named elliptic curves and KEMs) to allow on a wolfSSL context in order of preference. The list is a null_terminated, colon_delimited text string of group names, for example “P_384:P_256:X25519”. Call this function to set the key_exchange parameters used with TLS v1.3 connections (the function is compiled in only when HAVE_ECC, WOLFSSL_TLS13, and HAVE_SUPPORTED_CURVES are defined).
int	**wolfSSL_set1_groups_list**(WOLFSSL * ssl, const char * list)This function sets the list of key_exchange groups (named elliptic curves and KEMs) to allow on a wolfSSL session in order of preference. The list format and the set of recognized group names are identical to those documented for wolfSSL_CTX_set1_groups_list(); refer to that function for the full token tables and required build options.

	Name
int	**wolfSSL_CTX_set_groups**(WOLFSSL_CTX * ctx, int * groups, int count)This function sets the list of key_exchange groups (named elliptic curves and KEMs) to allow on a wolfSSL context in order of preference. The list is an array of named_group identifiers (see the table below) and count is the number of identifiers in the array. Use this function to set the key-exchange parameters used by TLS v1.3 connections; the order of the array becomes the local KeyShare preference order.
int	**wolfSSL_set_groups**(WOLFSSL * ssl, int * groups, int count)This function sets the list of key_exchange groups (named elliptic curves and KEMs) to allow on a wolfSSL session in order of preference. The array format and the set of recognized identifiers are identical to those documented for wolfSSL_CTX_set_groups(); refer to that function for the full identifier table and required build options.
int	**wolfSSL_CTX_set_max_early_data**(WOLFSSL_CTX * ctx, unsigned int sz)This function sets the maximum amount of early data that a TLS v1.3 client or server is willing to exchange using the wolfSSL context. Call this function to limit the amount of early data to process to mitigate replay attacks. Early data is protected by keys derived from those of the connection that the session ticket was sent and therefore will be the same every time a session ticket is used in resumption. The value is included in the session ticket for resumption. A server value of zero indicates no early data is to be sent by client using session tickets. A client value of zero indicates that the client will not send any early data. The default value is zero: per RFC 8446 Appendix E.5, TLS implementations “MUST NOT enable 0_RTT (either sending or accepting) unless specifically requested by the application.” Servers must call this function (or the per_SSL equivalent) with a non-zero value to opt in. It is recommended that the number of early data bytes be kept as low as practically possible in the application.

	Name
int	**wolfSSL_set_max_early_data**(WOLFSSL * ssl, unsigned int sz)This function sets the maximum amount of early data that a TLS v1.3 client or server is willing to exchange. Call this function to limit the amount of early data to process to mitigate replay attacks. Early data is protected by keys derived from those of the connection that the session ticket was sent and therefore will be the same every time a session ticket is used in resumption. The value is included in the session ticket for resumption. A server value of zero indicates no early data is to be sent by client using session tickets. A client value of zero indicates that the client will not send any early data. It is recommended that the number of early data bytes be kept as low as practically possible in the application.
int	**wolfSSL_CTX_no_early_data_fresh_start_check**(WOLFSSL_CTX * ctx)This function is called on the server to disable the RFC 8446 Section 8.2 fresh start protection. By default a freshly created context rejects early data, but not resumption, for session tickets minted before the context was created, since the anti_replay state for those tickets may not have survived a server restart. Only call this function when the anti_replay state (session cache or external cache) reliably survives server restarts.
void	**wolfSSL_CTX_set_psk_client_tls13_callback**(WOLFSSL_CTX * ctx, wc_psk_client_tls13_callback cb)This function sets the Pre_Shared Key (PSK) client side callback for TLS v1.3 connections. The callback is used to find a PSK identity and return its key and the name of the cipher to use for the handshake. The function sets the client_psk_tls13_cb member of the WOLFSSL_CTX structure.
void	**wolfSSL_set_psk_client_tls13_callback**(WOLFSSL * ssl, wc_psk_client_tls13_callback cb)This function sets the Pre_Shared Key (PSK) client side callback for TLS v1.3 connections. The callback is used to find a PSK identity and return its key and the name of the cipher to use for the handshake. The function sets the client_psk_tls13_cb member of the options field in WOLFSSL structure.

	Name
void	**wolfSSL_CTX_set_psk_server_tls13_callback**(WOLFSSL_CTX * ctx, wc_psk_server_tls13_callback cb)This function sets the Pre_Shared Key (PSK) server side callback for TLS v1.3 connections. The callback is used to find a PSK identity and return its key and the name of the cipher to use for the handshake. The function sets the server_psk_tls13_cb member of the WOLFSSL_CTX structure.
void	**wolfSSL_set_psk_server_tls13_callback**(WOLFSSL * ssl, wc_psk_server_tls13_callback cb)This function sets the Pre_Shared Key (PSK) server side callback for TLS v1.3 connections. The callback is used to find a PSK identity and return its key and the name of the cipher to use for the handshake. The function sets the server_psk_tls13_cb member of the options field in WOLFSSL structure.
int	**wolfSSL_CTX_set_cert_with_extern_psk**(WOLFSSL_CTX * ctx, int state)Enable or disable TLS 1.3 certificate authentication with external PSK (RFC 9973) on a context.
int	**wolfSSL_set_cert_with_extern_psk**(WOLFSSL * ssl, int state)Enable or disable TLS 1.3 certificate authentication with external PSK (RFC 9973) on a connection.
int	**wolfSSL_UseKeyShare**(WOLFSSL * ssl, word16 group)This function creates a key share entry from the group including generating a key pair. The KeyShare extension contains all the generated public keys for key exchange. If this function is called, then only the groups specified will be included. Call this function when a preferred group has been previously established for the server.
int	**wolfSSL_NoKeyShares**(WOLFSSL * ssl)This function is called to ensure no key shares are sent in the ClientHello. This will force the server to respond with a HelloRetryRequest if a key exchange is required in the handshake. Call this function when the expected key exchange group is not known and to avoid the generation of keys unnecessarily. Note that an extra round-trip will be required to complete the handshake when a key exchange is required.

	Name
WOLFSSL_METHOD *	**wolfTLSv1_3_server_method_ex**(void * heap)This function is used to indicate that the application is a server and will only support the TLS 1.3 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().
WOLFSSL_METHOD *	**wolfTLSv1_3_client_method_ex**(void * heap)This function is used to indicate that the application is a client and will only support the TLS 1.3 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().
WOLFSSL_METHOD *	**wolfTLSv1_3_server_method**(void)This function is used to indicate that the application is a server and will only support the TLS 1.3 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().
WOLFSSL_METHOD *	**wolfTLSv1_3_client_method**(void)This function is used to indicate that the application is a client and will only support the TLS 1.3 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().
WOLFSSL_METHOD *	**wolfTLSv1_3_method_ex**(void * heap)This function returns a WOLFSSL_METHOD similar to wolfTLSv1_3_client_method except that it is not determined which side yet (server/client).
WOLFSSL_METHOD *	**wolfTLSv1_3_method**(void)This function returns a WOLFSSL_METHOD similar to wolfTLSv1_3_client_method except that it is not determined which side yet (server/client).
int	**wolfSSL_CTX_set_client_cert_type**(WOLFSSL_CTX * ctx, const char * buf, int len)In case this function is called in a client side, set certificate types that can be sent to its peer. In case called in a server side, set certificate types that can be acceptable from its peer. Put cert types in the buffer with prioritised order. To reset the settings to default, pass NULL for the buffer or pass zero for len. By default, certificate type is only X509. In case both side intend to send or accept “Raw public key” cert, WOLFSSL_CERT_TYPE_RPK should be included in the buffer to set.

	Name
int	**wolfSSL_CTX_set_server_cert_type**(WOLFSSL_CTX * ctx, const char * buf, int len)In case this function is called in a server side, set certificate types that can be sent to its peer. In case called in a client side, set certificate types that can be acceptable from its peer. Put cert types in the buffer with prioritised order. To reset the settings to default, pass NULL for the buffer or pass zero for len. By default, certificate type is only X509. In case both side intend to send or accept “Raw public key” cert, WOLFSSL_CERT_TYPE_RPK should be included in the buffer to set.
int	**wolfSSL_set_client_cert_type**(WOLFSSL * ssl, const char * buf, int len)In case this function is called in a client side, set certificate types that can be sent to its peer. In case called in a server side, set certificate types that can be acceptable from its peer. Put cert types in the buffer with prioritised order. To reset the settings to default, pass NULL for the buffer or pass zero for len. By default, certificate type is only X509. In case both side intend to send or accept “Raw public key” cert, WOLFSSL_CERT_TYPE_RPK should be included in the buffer to set.
int	**wolfSSL_set_server_cert_type**(WOLFSSL * ssl, const char * buf, int len)In case this function is called in a server side, set certificate types that can be sent to its peer. In case called in a client side, set certificate types that can be acceptable from its peer. Put cert types in the buffer with prioritised order. To reset the settings to default, pass NULL for the buffer or pass zero for len. By default, certificate type is only X509. In case both side intend to send or accept “Raw public key” cert, WOLFSSL_CERT_TYPE_RPK should be included in the buffer to set.
int	**wolfSSL_CTX_clear_group_messages**(WOLFSSL_CTX * ctx)Disables handshake message grouping for the given WOLFSSL_CTX context.
int	**wolfSSL_clear_group_messages**(WOLFSSL * ssl)Disables handshake message grouping for the given WOLFSSL object.

	Name
int	**wolfSSL_CTX_set_expected_rpk**(WOLFSSL_CTX * ctx, const unsigned char * spki, unsigned int spkiSz)Pin a DER_encoded SubjectPublicKeyInfo that the peer is expected to present as a Raw Public Key (RFC 7250), establishing out_of_band trust on the WOLFSSL_CTX object. An unauthenticated RPK peer is otherwise rejected: when the peer is being authenticated (any verify mode other than WOLFSSL_VERIFY_NONE) the handshake fails closed unless the presented key matches a pin (or a verify callback accepts it). May be called more than once to pin several keys, up to WOLFSSL_MAX_RPK_PINS. The key is stored as its SHA_256 digest, so this API requires SHA_256. Pins are append_only _ there is no per_entry remove, but wolfSSL_CTX_clear_expected_rpk() empties the table so it can be repopulated.
int	**wolfSSL_set_expected_rpk**(WOLFSSL * ssl, const unsigned char * spki, unsigned int spkiSz)Pin a DER_encoded SubjectPublicKeyInfo that the peer is expected to present as a Raw Public Key (RFC 7250), establishing out_of_band trust on the WOLFSSL object. An unauthenticated RPK peer is otherwise rejected: when the peer is being authenticated (any verify mode other than WOLFSSL_VERIFY_NONE) the handshake fails closed unless the presented key matches a pin (or a verify callback accepts it). May be called more than once to pin several keys, up to WOLFSSL_MAX_RPK_PINS. The key is stored as its SHA_256 digest, so this API requires SHA_256. Pins are append_only _ there is no per_entry remove, but wolfSSL_clear_expected_rpk() empties the table so it can be repopulated.
int	**wolfSSL_CTX_clear_expected_rpk**(WOLFSSL_CTX * ctx)Remove all pinned expected peer Raw Public Keys (RFC 7250) from the WOLFSSL_CTX object, emptying the table so it can be repopulated _ for example across a peer key rotation, since the pinning APIs are otherwise append_only. Like other WOLFSSL_CTX setters this is not locked, so reconfigure the pins on a shared CTX while no handshakes are in flight (a WOLFSSL created with wolfSSL_new() copies the pin table by value at creation time).

	Name
int	**wolfSSL_clear_expected_rpk**(WOLFSSL * ssl)Remove all pinned expected peer Raw Public Keys (RFC 7250) from the WOLFSSL object, emptying the table so it can be repopulated - for example across a peer key rotation, since the pinning APIs are otherwise append-only.
int	**wolfSSL_get_scr_check_enabled**(const WOLFSSL * ssl)Gets the state of the secure renegotiation (SCR) check requirement.
int	**wolfSSL_set_scr_check_enabled**(WOLFSSL * ssl, byte enabled)Sets the state of the secure renegotiation (SCR) check requirement.
int	**wolfSSL_CTX_DisableExtendedMasterSecret**(WOLFSSL_CTX * ctx)Disables the TLS Extended Master Secret extension (RFC 7627) on the context: a client stops advertising it and a server ignores the peer's request, so a standard master secret is negotiated. A server also declines resumption of sessions or tickets that used EMS and does a full handshake instead. TLS 1.2 and earlier only. Requires HAVE_EXTENDED_MASTER.
int	**wolfSSL_DisableExtendedMasterSecret**(WOLFSSL * ssl)Disables the TLS Extended Master Secret extension (RFC 7627) on the SSL object: a client stops advertising it and a server ignores the peer's request, so a standard master secret is negotiated. A server also declines resumption of sessions or tickets that used EMS and does a full handshake instead. Call before the handshake starts, or after wolfSSL_clear. TLS 1.2 and earlier only. Requires HAVE_EXTENDED_MASTER.
int	**wolfSSL_CTX_EnableExtendedMasterSecret**(WOLFSSL_CTX * ctx)Re_enables the TLS Extended Master Secret extension (RFC 7627) on the context (the default): EMS is used when the peer supports it but is not mandatory. Undoes a previous disable or require. Requires HAVE_EXTENDED_MASTER.
int	**wolfSSL_EnableExtendedMasterSecret**(WOLFSSL * ssl)Re_enables the TLS Extended Master Secret extension (RFC 7627) on the SSL object (the default): EMS is used when the peer supports it but is not mandatory. Undoes a previous disable or require. Call before the handshake starts, or after wolfSSL_clear. Requires HAVE_EXTENDED_MASTER.

	Name
int	**wolfSSL_CTX_RequireExtendedMasterSecret**(WOLFSSL_CTX * ctx)Makes the TLS Extended Master Secret extension (RFC 7627) mandatory on the context: if it is not negotiated, the connection is aborted with EXT_MASTER_SECRET_NEEDED_E. A client advertises the extension even after a previous disable. TLS 1.2 and earlier only. Requires HAVE_EXTENDED_MASTER.
int	**wolfSSL_RequireExtendedMasterSecret**(WOLFSSL * ssl)Makes the TLS Extended Master Secret extension (RFC 7627) mandatory on the SSL object: if it is not negotiated, including on resumption, the connection is aborted with EXT_MASTER_SECRET_NEEDED_E. A client advertises the extension even after a previous disable. Call before the handshake starts, or after wolfSSL_clear. TLS 1.2 and earlier only. Requires HAVE_EXTENDED_MASTER.
void *	**wolfSSL_GetCookieCtx**(WOLFSSL * ssl)This function returns the IOCB_CookieCtx member of the WOLFSSL structure.
int	**wolfSSL_SetIO_ISOTP**(WOLFSSL * ssl, isotp_wolfssl_ctx * ctx, can_recv_fn recv_fn, can_send_fn send_fn, can_delay_fn delay_fn, word32 receive_delay, char * receive_buffer, int receive_buffer_size, void * arg)This function sets up the ISO-TP context if wolfSSL, for use when wolfSSL is compiled with WOLFSSL_ISOTP.
void	**wolfSSL_SSLDisableRead**(WOLFSSL * ssl)This function disables reading from the IO layer.
void	**wolfSSL_SSLEnableRead**(WOLFSSL * ssl)This function enables reading from the IO layer. Reading is enabled by default and should be used to undo wolfSSL_SSLDisableRead();.

A.6.3 Types Documentation

```
typedef void(* WOLFSSL_TLSEXT_DEBUG_CB) (WOLFSSL *ssl, int client_server, int
   type, const byte *data, int len, void *arg);
```

Callback type for the TLS extension debug callback.

Parameters:

- **ssl** The WOLFSSL object receiving the extension.
- **client_server** 1 if the WOLFSSL object is a client, 0 if a server.
- **type** The extension type, e.g. TLSX_SERVER_NAME.
- **data** The raw extension content (data after the 2-byte length).
- **len** Length of the extension content in bytes.
- **arg** The argument set with wolfSSL_set_tlsext_debug_arg().

Invoked once for every TLS extension received during the handshake, in wire order, before the extension is processed.

Note that, unlike OpenSSL 3.x, the callback also reports unknown (unregistered) extension types.

A.6.4 Functions Documentation

A.6.3.1 typedef WOLFSSL_TLSEXT_DEBUG_CB

```
WOLFSSL_METHOD * wolfSSLv23_method(
    void
)
```

This function returns a WOLFSSL_METHOD similar to wolfSSLv23_client_method except that it is not determined which side yet (server/client).

Parameters:

- **none** No parameters.

See:

- wolfSSL_new
- wolfSSL_free

Return:

- WOLFSSL_METHOD* On successful creations returns a WOLFSSL_METHOD pointer
- NULL Null if memory allocation error or failure to create method

Example

```
WOLFSSL* ctx;
ctx  = wolfSSL_CTX_new(wolfSSLv23_method());
// check ret value
```

```
WOLFSSL_METHOD * wolfSSLv3_server_method(
    void
)
```

The wolfSSLv3_server_method() function is used to indicate that the application is a server and will only support the SSL 3.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Parameters:

- **none** No parameters.

See:

- wolfTLSv1_server_method
- wolfTLSv1_1_server_method
- wolfTLSv1_2_server_method
- wolfTLSv1_3_server_method
- wolfDTLSv1_server_method
- wolfSSLv23_server_method
- wolfSSL_CTX_new

Return:

-
  - If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfSSLv3_server_method();
if (method == NULL) {
    unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

```
WOLFSSL_METHOD * wolfSSLv3_client_method(
    void
)
```

The wolfSSLv3_client_method() function is used to indicate that the application is a client and will only support the SSL 3.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Parameters:

- **none** No parameters.

See:

- wolfTLSv1_client_method
- wolfTLSv1_1_client_method

- wolfTLSv1_2_client_method
- wolfTLSv1_3_client_method
- wolfDTLSv1_client_method
- wolfSSLv23_client_method
- wolfSSL_CTX_new

Return:

-
  - If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfSSLv3_client_method();
if (method == NULL) {
    unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

```
WOLFSSL_METHOD * wolfTLSv1_server_method(
    void
)
```

The wolfTLSv1_server_method() function is used to indicate that the application is a server and will only support the TLS 1.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Parameters:

- **none** No parameters.

See:

- wolfSSLv3_server_method
- wolfTLSv1_1_server_method
- wolfTLSv1_2_server_method
- wolfTLSv1_3_server_method
- wolfDTLSv1_server_method
- wolfSSLv23_server_method
- wolfSSL_CTX_new

Return:

-
  - If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_server_method();
if (method == NULL) {
    unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

```
WOLFSSL_METHOD * wolfTLSv1_client_method(
    void
)
```

The wolfTLSv1_client_method() function is used to indicate that the application is a client and will only support the TLS 1.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Parameters:

- **none** No parameters.

See:

- wolfSSLv3_client_method
- wolfTLSv1_1_client_method
- wolfTLSv1_2_client_method
- wolfTLSv1_3_client_method
- wolfDTLSv1_client_method
- wolfSSLv23_client_method
- wolfSSL_CTX_new

Return:

-
  - If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.

- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_client_method();
if (method == NULL) {
    unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

```
WOLFSSL_METHOD * wolfTLSv1_1_server_method(
    void
)
```

The wolfTLSv1_1_server_method() function is used to indicate that the application is a server and will only support the TLS 1.1 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Parameters:

- **none** No parameters.

See:

- wolfSSLv3_server_method
- wolfTLSv1_server_method
- wolfTLSv1_2_server_method
- wolfTLSv1_3_server_method
- wolfDTLSv1_server_method
- wolfSSLv23_server_method
- wolfSSL_CTX_new

Return:

-
  - If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_1_server_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

```
WOLFSSL_METHOD * wolfTLSv1_1_client_method(
    void
)
```

The wolfTLSv1_1_client_method() function is used to indicate that the application is a client and will only support the TLS 1.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Parameters:

- **none** No parameters.

See:

- wolfSSLv3_client_method
- wolfTLSv1_client_method
- wolfTLSv1_2_client_method
- wolfTLSv1_3_client_method
- wolfDTLSv1_client_method
- wolfSSLv23_client_method
- wolfSSL_CTX_new

Return:

-
  - If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;
```

```
method = wolfTLSv1_1_client_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

```
WOLFSSL_METHOD * wolfTLSv1_2_server_method(
    void
)
```

The wolfTLSv1_2_server_method() function is used to indicate that the application is a server and will only support the TLS 1.2 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Parameters:

- **none** No parameters.

See:

- wolfSSLv3_server_method
- wolfTLSv1_server_method
- wolfTLSv1_1_server_method
- wolfTLSv1_3_server_method
- wolfDTLSv1_server_method
- wolfSSLv23_server_method
- wolfSSL_CTX_new

Return:

-
  - If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_2_server_method();
if (method == NULL) {
    // unable to get method
}
```

```
ctx = wolfSSL_CTX_new(method);
...
```

```
WOLFSSL_METHOD * wolfTLSv1_2_client_method(
    void
)
```

The wolfTLSv1_2_client_method() function is used to indicate that the application is a client and will only support the TLS 1.2 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Parameters:

- **none** No parameters.

See:

- wolfSSLv3_client_method
- wolfTLSv1_client_method
- wolfTLSv1_1_client_method
- wolfTLSv1_3_client_method
- wolfDTLSv1_client_method
- wolfSSLv23_client_method
- wolfSSL_CTX_new

Return:

-
  - If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_2_client_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

```
WOLFSSL_METHOD * wolfDTLSv1_client_method(
    void
)
```

The wolfDTLSv1_client_method() function is used to indicate that the application is a client and will only support the DTLS 1.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). This function is only available when wolfSSL has been compiled with DTLS support (–enable-dtls, or by defining wolfSSL_DTLS).

Parameters:

- **none** No parameters.

See:

- wolfSSLv3_client_method
- wolfTLSv1_client_method
- wolfTLSv1_1_client_method
- wolfTLSv1_2_client_method
- wolfTLSv1_3_client_method
- wolfSSLv23_client_method
- wolfSSL_CTX_new

Return:

-
  - If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLSv1_client_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

```
WOLFSSL_METHOD * wolfDTLSv1_server_method(
    void
)
```

The wolfDTLSv1_server_method() function is used to indicate that the application is a server and will only support the DTLS 1.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). This function is only available when wolfSSL has been compiled with DTLS support (–enable-dtls, or by defining wolfSSL_DTLS).

Parameters:

- **none** No parameters.

See:

- wolfSSLv3_server_method
- wolfTLSv1_server_method
- wolfTLSv1_1_server_method
- wolfTLSv1_2_server_method
- wolfTLSv1_3_server_method
- wolfSSLv23_server_method
- wolfSSL_CTX_new

Return:

-
  - If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLSv1_server_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

```
WOLFSSL_METHOD * wolfDTLSv1_3_server_method(
    void
)
```

The wolfDTLSv1_3_server_method() function is used to indicate that the application is a server and will only support the DTLS 1.3 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). This function is only available when wolfSSL has been compiled with DTLSv1.3 support (–enable-dtls13, or by defining wolfSSL_DTLS13).

Parameters:

- **none** No parameters.

See: wolfDTLSv1_3_client_method

Return:

-
  - If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLSv1_3_server_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

```
WOLFSSL_METHOD * wolfDTLSv1_3_client_method(
    void
)
```

The wolfDTLSv1_3_client_method() function is used to indicate that the application is a client and will only support the DTLS 1.3 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). This function is only available when wolfSSL has been compiled with DTLSv1.3 support (–enable-dtls13, or by defining wolfSSL_DTLS13).

Parameters:

- **none** No parameters.

See: wolfDTLSv1_3_server_method

Return:

-
  - If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLSv1_3_client_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

```
WOLFSSL_METHOD * wolfDTLS_server_method(
    void
)
```

The wolfDTLS_server_method() function is used to indicate that the application is a server and will support the highest version of DTLS available and all the version up to the minimum version allowed. The default minimum version allowed is based on the define WOLFSSL_MIN_DTLS_DOWNGRADE and can be changed at runtime using wolfSSL_SetMinVersion(). This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). This function is only available when wolfSSL has been compiled with DTLS support (–enable-dtls, or by defining wolfSSL_DTLS).

Parameters:

- **none** No parameters.

See:

- wolfDTLS_client_method
- wolfSSL_SetMinVersion

Return:

-
  - If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLS_server_method();
if (method == NULL) {
```

```
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

```
WOLFSSL_METHOD * wolfDTLS_client_method(
    void
)
```

The wolfDTLS_client_method() function is used to indicate that the application is a client and will support the highest version of DTLS available and all the version up to the minimum version allowed. The default minimum version allowed is based on the define WOLFSSL_MIN_DTLS_DOWNGRADE and can be changed at runtime using wolfSSL_SetMinVersion(). This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). This function is only available when wolfSSL has been compiled with DTLS support (–enable-dtls, or by defining wolfSSL_DTLS).

Parameters:

- **none** No parameters.

See:

- wolfDTLS_server_method
- wolfSSL_SetMinVersion

Return:

-
  - If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLS_client_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

```
int wolfSSL_use_old_poly(
    WOLFSSL * ssl,
    int value
)
```

Since there is some differences between the first release and newer versions of chacha-poly AEAD construction we have added an option to communicate with servers/clients using the older version. By default wolfSSL uses the new version.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **value** whether or not to use the older version of setting up the information for poly1305. Passing a flag value of 1 indicates yes use the old poly AEAD, to switch back to using the new version pass a flag value of 0.

See: none

Return: 0 upon success

Example

```
int ret = 0;
WOLFSSL* ssl;
...

ret = wolfSSL_use_old_poly(ssl, 1);
if (ret != 0) {
    // failed to set poly1305 AEAD version
}
```

```
int wolfSSL_CTX_trust_peer_cert(
    WOLFSSL_CTX * ctx,
    const char * file,
    int type
)
```

This function loads a certificate to use for verifying a peer when performing a TLS/SSL handshake. The peer certificate sent during the handshake is compared by using the SKID when available and the signature. If these two things do not match then any loaded CAs are used. Feature is enabled by defining the macro WOLFSSL_TRUST_PEER_CERT. Please see the examples for proper usage.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **file** pointer to name of the file containing certificates
- **type** type of certificate being loaded ie SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CTX_load_verify_buffer

- wolfSSL_CTX_use_certificate_file
- wolfSSL_CTX_use_PrivateKey_file
- wolfSSL_CTX_use_certificate_chain_file
- wolfSSL_CTX_trust_peer_buffer
- wolfSSL_CTX_Unload_trust_peers
- wolfSSL_use_certificate_file
- wolfSSL_use_PrivateKey_file
- wolfSSL_use_certificate_chain_file

Return:

- SSL_SUCCES upon success.
- SSL_FAILURE will be returned if ctx is NULL, or if both file and type are invalid.
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
...

ret = wolfSSL_CTX_trust_peer_cert(ctx, "./peer-cert.pem",
SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading trusted peer cert
}
...
```

```
long wolfSSL_CTX_get_verify_depth(
    WOLFSSL_CTX * ctx
)
```

This function gets the certificate chaining depth using the CTX structure.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

See:

- wolfSSL_CTX_use_certificate_chain_file
- wolfSSL_get_verify_depth

Return:

- MAX_CHAIN_DEPTH returned if the CTX struct is not NULL. The constant representation of the max certificate chain peer depth.

- BAD_FUNC_ARG returned if the CTX structure is NULL.

Example

```
WOLFSSL_METHOD method; // protocol method
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(method);
…
long ret = wolfSSL_CTX_get_verify_depth(ctx);

if(ret == EXPECTED){
    //  You have the expected value
} else {
    //  Handle an unexpected depth
}
```

```
WOLFSSL_CTX * wolfSSL_CTX_new(
    WOLFSSL_METHOD *
)
```

This function creates a new SSL context, taking a desired SSL/TLS protocol method for input.

Parameters:

- **method** pointer to the desired WOLFSSL_METHOD to use for the SSL context. This is created using one of the wolfSSLvXX_XXXX_method() functions to specify SSL/TLS/DTLS protocol level. This function frees the passed in WOLFSSL_METHOD struct on failure.

See: wolfSSL_new

Return:

- pointer If successful the call will return a pointer to the newly-created WOLFSSL_CTX.
- NULL upon failure.

Example

```
WOLFSSL_CTX*    ctx    = 0;
WOLFSSL_METHOD* method = 0;

method = wolfSSLv3_client_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // context creation failed
}
```

```
WOLFSSL * wolfSSL_new(
    WOLFSSL_CTX *
)
```

This function creates a new SSL session, taking an already created SSL context as input.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().

See: wolfSSL_CTX_new

Return:

-
  - If successful the call will return a pointer to the newly-created wolfSSL structure.
- NULL Upon failure.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL*     ssl = NULL;
WOLFSSL_CTX* ctx = 0;

ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // context creation failed
}

ssl = wolfSSL_new(ctx);
if (ssl == NULL) {
    // SSL object creation failed
}
```

```
int wolfSSL_set_fd(
    WOLFSSL * ssl,
    int fd
)
```

This function assigns a file descriptor (fd) as the input/output facility for the SSL connection. Typically this will be a socket file descriptor.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **fd** file descriptor to use with SSL/TLS connection.

See:

- wolfSSL_CTX_SetIOSend
- wolfSSL_CTX_SetIORecv
- wolfSSL_SetIOReadCtx
- wolfSSL_SetIOWriteCtx

Return:

- SSL_SUCCESS upon success.
- BAD_FUNC_ARG upon failure.

Example

```
int sockfd;
WOLFSSL* ssl = 0;
...

ret = wolfSSL_set_fd(ssl, sockfd);
if (ret != SSL_SUCCESS) {
    // failed to set SSL file descriptor
}
```

```
int wolfSSL_set_dtls_fd_connected(
    WOLFSSL * ssl,
    int fd
)
```

This function assigns a file descriptor (fd) as the input/output facility for the SSL connection. Typically this will be a socket file descriptor. This is a DTLS specific API because it marks that the socket is connected. recvfrom and sendto calls on this fd will have the addr and addr_len parameters set to NULL.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **fd** file descriptor to use with SSL/TLS connection.

See:

- wolfSSL_CTX_SetIOSend
- wolfSSL_CTX_SetIORecv
- wolfSSL_SetIOReadCtx
- wolfSSL_SetIOWriteCtx
- wolfDTLS_SetChGoodCb

Return:

- SSL_SUCCESS upon success.
- BAD_FUNC_ARG upon failure.

Example

```
int sockfd;
WOLFSSL* ssl = 0;
...
if (connect(sockfd, peer_addr, peer_addr_len) != 0) {
    // handle connect error
}
...
ret = wolfSSL_set_dtls_fd_connected(ssl, sockfd);
if (ret != SSL_SUCCESS) {
    // failed to set SSL file descriptor
}
```

```
int wolfDTLS_SetChGoodCb(
    WOLFSSL * ssl,
    ClientHelloGoodCb cb,
    void * user_ctx
)
```

Allows setting a callback for a correctly processed and verified DTLS client hello. When using a cookie exchange mechanism (either the HelloVerifyRequest in DTLS 1.2 or the HelloRetryRequest with a cookie extension in DTLS 1.3) this callback is called after the cookie exchange has succeeded. This is useful to use one WOLFSSL object as the listener for new connections and being able to isolate the WOLFSSL object once the ClientHello is verified (either through a cookie exchange or just checking if the ClientHello had the correct format). With DTLS cookies disabled, the callback is invoked once per ClientHello, only after complete successful ClientHello processing, not on its first fragment. Retries, retransmitted fragments and a second ClientHello after a key-share HelloRetryRequest do not repeat the notification. The object is already stateful before input is read, but the peer's return-routability has not been verified. Returning 0 accepts the ClientHello and the handshake continues. Any negative return stops the accept call with that error and leaves the accept state machine at the ClientHello, so nothing is sent to the peer and the next accept call asks the callback again instead of continuing the handshake. All negative returns behave this way; the value only decides what wolfSSL_get_error() reports, so what it means is the application's choice. A callback that is not ready to decide can return WANT_READ or WANT_WRITE and be asked again as many times as it needs, and one that refuses the peer returns its own error and keeps the server silent by refusing again on every later accept call. The cookie mode cannot be changed from the callback; see wolfSSL_disable_cookie(). DTLS 1.2: https://datatracker.ietf.org/doc/html/rfc6347#section_4.2.1 DTLS 1.3: https://www.rfc_editor.org/rfc/rfc8446#section_4.2.2.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **cb** ClientHelloGoodCb callback function pointer.
- **user_ctx** pointer to user context to be passed to callback.

See: wolfSSL_set_dtls_fd_connected

Return:

- SSL_SUCCESS upon success.

- BAD_FUNC_ARG upon failure.

Example

```
// Called when we have verified a connection
static int chGoodCb(WOLFSSL* ssl, void* arg)
{
    // setup peer and file descriptors

}

if (wolfDTLS_SetChGoodCb(ssl, chGoodCb, NULL) != WOLFSSL_SUCCESS) {
     // error setting callback
}
```

```
void wolfSSL_set_using_nonblock(
    WOLFSSL * ssl,
    int nonblock
)
```

This function informs the WOLFSSL object that the underlying I/O is non-blocking. After an application creates a WOLFSSL object, if it will be used with a non-blocking socket, call wolfSSL_set_using_nonblock() on it. This lets the WOLFSSL object know that receiving EWOULDBLOCK means that the recvfrom call would block rather than that it timed out.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **nonblock** value used to set non-blocking flag on WOLFSSL object. Use 1 to specify non-blocking, otherwise 0.

See:

- wolfSSL_get_using_nonblock
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls_get_current_timeout

Return: none No return.

Example

```
WOLFSSL* ssl = 0;
...
wolfSSL_set_using_nonblock(ssl, 1);
```

```
int wolfSSL_disable_cookie(
    WOLFSSL * ssl
)
```

Disable server cookies for DTLS 1.2, DTLS 1.3 and TLS 1.3, including DTLS 1.2 fallback from DTLS 1.3. Cookies are enabled by default for DTLS. For DTLS, ordinary wolfSSL_accept() or wolfSSL_accept_TLSv13() commits it to stateful processing before the first read, even if a nonblocking accept has no input available. The application must isolate/demultiplex the peer before accepting; disabling cookies removes return_routability verification and exposes the server to DoS/amplification attacks. wolfDTLS_accept_stateless() cannot be used.

Parameters:

- **ssl** DTLS or TLS 1.3 server session created with wolfSSL_new(). A general-purpose session that has not chosen a side yet is accepted: the policy set here is what the promotion to the server side finds.

See:

- wolfSSL_enable_cookie
- wolfDTLS_SetChGoodCb

Return:

- WOLFSSL_SUCCESS on success (including an unchanged mode).
- BAD_FUNC_ARG if ssl is NULL or uses an unsupported protocol (TLS 1.2).
- SIDE_ERROR if ssl is a client.
- BAD_STATE_E if the DTLS handshake already committed to stateful processing.

Primary and secondary cookie secrets for the applicable protocols are securely erased and freed.

Without cookies a DTLS 1.3 server can process a first ClientHello that arrives fragmented, but only when ClientHello fragment reassembly is available: wolfSSL must be built with WOLFSSL_DTLS_CH_FRAG (–enable-dtls-frag-ch, turned on automatically with ML-KEM and DTLS 1.3) and it must be enabled on the object with wolfSSL_dtls13_allow_ch_frag(), which is done by default only in builds with ML-KEM. Otherwise the fragments are dropped and the handshake does not progress.

The DTLS cookie mode is fixed once the handshake commits to stateful processing, which the accept functions do before the first read and stateless processing does when a cookie verifies. After that this call fails instead of reporting a change that cannot take effect, so it cannot be used from a ClientHello good callback.

```
int wolfSSL_enable_cookie(
    WOLFSSL * ssl
)
```

Enable server cookies for DTLS 1.2, DTLS 1.3 and TLS 1.3. Like wolfSSL_disable_cookie(), this does not change stateful processing. It can undo a disable before accept begins, including wolfSSL_disable_hrr_cookie(). Missing primary cookie secrets are randomly generated immediately; existing primary and secondary secrets are preserved.

Parameters:

- **ssl** DTLS or TLS 1.3 server session created with wolfSSL_new(). A general-purpose session that has not chosen a side yet is accepted: the policy set here is what the promotion to the server side finds.

See: wolfSSL_disable_cookie

Return:

- WOLFSSL_SUCCESS on success (including an unchanged mode).
- BAD_FUNC_ARG if ssl is NULL or uses an unsupported protocol (TLS 1.2).
- SIDE_ERROR if ssl is a client.
- BAD_STATE_E if the DTLS handshake already committed to stateful processing, as for wolfSSL_disable_cookie(), or if a missing secret must be generated and the session's RNG has been released (as happens when its handshake resources are freed).
- MEMORY_ERROR if secret allocation fails, or another negative error if random secret generation fails. A failure is all or nothing: no secret is installed and the cookie policy is left as it was found, even when the session needs both a DTLS 1.2 and a DTLS 1.3 secret.

```
void wolfSSL_CTX_free(
    WOLFSSL_CTX * ctx
)
```

This function frees an allocated WOLFSSL_CTX object. This function decrements the CTX reference count and only frees the context when the reference count has reached 0.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().

See:

- wolfSSL_CTX_new
- wolfSSL_new
- wolfSSL_free

Return: none No return.

Example

```
WOLFSSL_CTX* ctx = 0;
...
wolfSSL_CTX_free(ctx);
```

```
void wolfSSL_free(
    WOLFSSL * ssl
)
```

This function frees an allocated wolfSSL object.

Parameters:

- **ssl** pointer to the SSL object, created with wolfSSL_new().

See:

- wolfSSL_CTX_new
- wolfSSL_new
- wolfSSL_CTX_free

Return: none No return.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL* ssl = 0;
...
wolfSSL_free(ssl);
```

```
int wolfSSL_set_session(
    WOLFSSL * ssl,
    WOLFSSL_SESSION * session
)
```

This function sets the session to be used when the SSL object, ssl, is used to establish a SSL/TLS connection. For session resumption, before calling wolfSSL_shutdown() with your session object, an application should save the session ID from the object with a call to wolfSSL_get1_session(), which returns a pointer to the session. Later, the application should create a new WOLFSSL object and assign the saved session with wolfSSL_set_session(). At this point, the application may call wolfSSL_connect() and wolfSSL will try to resume the session. The wolfSSL server code allows session resumption by default. The object returned by wolfSSL_get1_session() needs to be freed after the application is done with it by calling wolfSSL_SESSION_free() on it.

Parameters:

- **ssl** pointer to the SSL object, created with wolfSSL_new().
- **session** pointer to the WOLFSSL_SESSION used to set the session for ssl.

See: wolfSSL_get1_session

Return:

- SSL_SUCCESS will be returned upon successfully setting the session.
- SSL_FAILURE will be returned on failure. This could be caused by the session cache being disabled, the session having timed out, or an EMS session being declined because EMS is disabled.
- When OPENSSL_EXTRA and WOLFSSL_ERROR_CODE_OPENSSL are defined, SSL_SUCCESS will be returned even if the session has timed out.

Example

```
int ret;
WOLFSSL* ssl;
WOLFSSL_SESSION* session;
```

```
...
session = wolfSSL_get1_session(ssl);
if (session == NULL) {
    // failed to get session object from ssl object
}
...
ret = wolfSSL_set_session(ssl, session);
if (ret != SSL_SUCCESS) {
    // failed to set the SSL session
}
wolfSSL_SESSION_free(session);
...
```

```
void wolfSSL_CTX_set_verify(
    WOLFSSL_CTX * ctx,
    int mode,
    VerifyCallback verify_callback
)
```

This function sets the verification method for remote peers and also allows a verify callback to be registered with the SSL context. The verify callback will be called only when a verification failure has occurred. If no verify callback is desired, the NULL pointer can be used for verify_callback. The verification mode of peer certificates is a logically OR'd list of flags. The possible flag values include: SSL_VERIFY_NONE Client mode: the client will not verify the certificate received from the server and the handshake will continue as normal. Server mode: the server will not send a certificate request to the client. As such, client verification will not be enabled. SSL_VERIFY_PEER Client mode: the client will verify the certificate received from the server during the handshake. This is turned on by default in wolfSSL, therefore, using this option has no effect. Server mode: the server will send a certificate request to the client and verify the client certificate received. SSL_VERIFY_FAIL_IF_NO_PEER_CERT Client mode: no effect when used on the client side. Server mode: the verification will fail on the server side if the client fails to send a certificate when requested to do so (when using SSL_VERIFY_PEER on the SSL server). SSL_VERIFY_FAIL_EXCEPT_PSK Client mode: no effect when used on the client side. Server mode: the verification is the same as SSL_VERIFY_FAIL_IF_NO_PEER_CERT except in the case of a PSK connection. If a PSK connection is being made then the connection will go through without a peer cert. SSL_VERIFY_CLIENT_ONCE Accepted and ignored. It is present so that OpenSSL-derived code compiles unchanged; wolfSSL does not act on it, does not store it, and wolfSSL_get_verify_mode() / wolfSSL_CTX_get_verify_mode() will not report it back. Including it in mode has no effect on any of the other flags.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **mode** flags indicating verification mode for peer's cert.
- **verify_callback** callback to be called when verification fails. If no callback is desired, the NULL pointer can be used for verify_callback.

See: wolfSSL_set_verify

Return: none No return.

Example

```
WOLFSSL_CTX*    ctx    = 0;
...
wolfSSL_CTX_set_verify(ctx, (WOLFSSL_VERIFY_PEER |
                       WOLFSSL_VERIFY_FAIL_IF_NO_PEER_CERT), NULL);
```

```
void wolfSSL_set_verify(
    WOLFSSL * ssl,
    int mode,
    VerifyCallback verify_callback
)
```

This function sets the verification method for remote peers and also allows a verify callback to be registered with the SSL session. The verify callback will be called only when a verification failure has occurred. If no verify callback is desired, the NULL pointer can be used for verify_callback. The verification mode of peer certificates is a logically OR'd list of flags. The possible flag values include: SSL_VERIFY_NONE Client mode: the client will not verify the certificate received from the server and the handshake will continue as normal. Server mode: the server will not send a certificate request to the client. As such, client verification will not be enabled. SSL_VERIFY_PEER Client mode: the client will verify the certificate received from the server during the handshake. This is turned on by default in wolfSSL, therefore, using this option has no effect. Server mode: the server will send a certificate request to the client and verify the client certificate received. SSL_VERIFY_FAIL_IF_NO_PEER_CERT Client mode: no effect when used on the client side. Server mode: the verification will fail on the server side if the client fails to send a certificate when requested to do so (when using SSL_VERIFY_PEER on the SSL server). SSL_VERIFY_FAIL_EXCEPT_PSK Client mode: no effect when used on the client side. Server mode: the verification is the same as SSL_VERIFY_FAIL_IF_NO_PEER_CERT except in the case of a PSK connection. If a PSK connection is being made then the connection will go through without a peer cert. SSL_VERIFY_CLIENT_ONCE Accepted and ignored. It is present so that OpenSSL-derived code compiles unchanged; wolfSSL does not act on it, does not store it, and wolfSSL_get_verify_mode() / wolfSSL_CTX_get_verify_mode() will not report it back. Including it in mode has no effect on any of the other flags.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **mode** flags indicating verification mode for peer's cert.
- **verify_callback** callback to be called when verification fails. If no callback is desired, the NULL pointer can be used for verify_callback.

See: wolfSSL_CTX_set_verify

Return: none No return.

Example

```
WOLFSSL* ssl = 0;
...
wolfSSL_set_verify(ssl, SSL_VERIFY_PEER | SSL_VERIFY_FAIL_IF_NO_PEER_CERT, 0);
```

```
long wolfSSL_CTX_set_session_cache_mode(
    WOLFSSL_CTX * ctx,
```

```
    long mode
)
```

This function enables or disables SSL session caching. Behavior depends on the value used for mode. The following values for mode are available: SSL_SESS_CACHE_OFF- disable session caching. Session caching is turned on by default. SSL_SESS_CACHE_NO_AUTO_CLEAR - Disable auto-flushing of the session cache. Auto-flushing is turned on by default.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **mode** modifier used to change behavior of the session cache.

See:

- wolfSSL_flush_sessions
- wolfSSL_get1_session
- wolfSSL_set_session
- wolfSSL_get_sessionID
- wolfSSL_CTX_set_timeout

Return: SSL_SUCCESS will be returned upon success.

Example

```
WOLFSSL_CTX* ctx = 0;
...
ret = wolfSSL_CTX_set_session_cache_mode(ctx, SSL_SESS_CACHE_OFF);
if (ret != SSL_SUCCESS) {
    // failed to turn SSL session caching off
}
```

```
void wolfSSL_CTX_sess_set_get_cb(
    WOLFSSL_CTX * ctx,
    WOLFSSL_SESSION *(*)(WOLFSSL *ssl, const unsigned char *, int, int *) f
)
```

This function registers the callback wolfSSL uses to look a session up in an application managed (external) session cache. It is called on the server side when a peer offers a session id to resume, before the internal cache is consulted. Compatibility layer equivalent of SSL_CTX_sess_set_get_cb().

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **f** the lookup callback, or NULL to remove a previously registered one. It is passed the WOLFSSL object being negotiated, the session id to look up and its length, and a pointer to a copy flag. It returns the matching WOLFSSL_SESSION, or NULL when the cache holds no session for that id. wolfSSL sets the copy flag to 1 before the call. Left at 1, the callback keeps its reference and wolfSSL never releases it, so the session must stay valid while wolfSSL processes the lookup. For TLS 1.3 stateful ticket resumption, which keeps the session longer, wolfSSL takes a reference of its own and releases it when done; a session it cannot take a reference on is then treated as no

session found. Set to 0, the callback hands its reference to wolfSSL, which releases it once done with the session.

See:

- wolfSSL_CTX_set_session_cache_mode
- wolfSSL_get1_session
- wolfSSL_SESSION_free
- wolfSSL_flush_sessions

Return: none No return.

Example

```
WOLFSSL_SESSION* myGetSession(WOLFSSL* ssl, const unsigned char* id,
                              int idLen, int* copy)
{
    (void)ssl;
    // The cache keeps its session, so leave copy alone.
    return myCacheLookup(id, idLen);
}
...
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(method);
wolfSSL_CTX_sess_set_get_cb(ctx, myGetSession);
```

```
int wolfSSL_CTX_memrestore_cert_cache(
    WOLFSSL_CTX * ctx,
    const void * mem,
    int sz
)
```

This function restores the certificate cache from memory.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **mem** a void pointer with a value that will be restored to the certificate cache.
- **sz** an int type that represents the size of the mem parameter.

See: CM_MemRestoreCertCache

Return:

- SSL_SUCCESS returned if the function and subroutines executed without an error.
- BAD_FUNC_ARG returned if the ctx or mem parameters are NULL or if the sz parameter is less than or equal to zero.
- BUFFER_E returned if the cert cache memory buffer is too small.
- CACHE_MATCH_ERROR returned if there was a cert cache header mismatch.
- BAD_MUTEX_E returned if the lock mutex on failed.

Example

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new( protocol method );
WOLFSSL* ssl = WOLFSSL_new(ctx);
void* mem;
int sz = (*int) sizeof(mem);
…
if(wolfSSL_CTX_memrestore_cert_cache(ssl->ctx, mem, sz)){
    // The success case
}
```

```
int wolfSSL_CTX_set_cipher_list(
    WOLFSSL_CTX * ctx,
    const char * list
)
```

This function sets the cipher suite list for a given WOLFSSL_CTX. The list becomes the default cipher suite list for any new WOLFSSL sessions created from the context, and the order in the string is the local preference order from highest to lowest. Each call replaces the previous list. The list is a null-terminated, colon-delimited text string of suite names and/or OpenSSL-style group keywords, for example “TLS13-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:DHE-RSA-AES256-SHA256”.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **list** null-terminated text string and a colon-delimited list of cipher suites and/or keywords to use with the specified SSL context.

See:

- wolfSSL_set_cipher_list
- wolfSSL_get_ciphers
- wolfSSL_get_cipher_list
- wolfSSL_CTX_new

Return:

- WOLFSSL_SUCCESS will be returned upon successful function completion.
- WOLFSSL_FAILURE will be returned on failure.

Each token in the colon-delimited list is one of the following:

1. A specific cipher suite name. wolfSSL accepts both its own short name and the IANA name (when WOLFSSL_NO_ERROR_STRINGS is not defined). For example “ECDHE-RSA-AES128-GCM-SHA256” and “TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256” are equivalent. The complete authoritative list of accepted names lives in the cipher_names[] array in src/internal.c; programs may also enumerate the suites compiled into the current build at runtime via wolfSSL_get_ciphers().
2. An OpenSSL-compatible keyword that selects a family of suites or toggles a class on/off (see the keyword table below). Some keywords require OPENSSL_EXTRA or OPENSSL_ALL to be defined.

3. A negated keyword “!keyword” to disallow a class (requires OPENSSL_EXTRA or OPENSSL_ALL). For example “HIGH:!aNULL”.

A “+” operator (e.g. “ECDHE+AESGCM”) is recognized only to extract the leading public-key family token (“ECDHE”, “RSA” or “DHE”); trailing parts after “+” are ignored by wolfSSL. To intersect with a specific cipher use the explicit suite name instead.

OpenSSL-compatible group keywords:

Keyword	Effect	Required build option
DEFAULT / ALL	Include all built suites; “ALL” also enables anonymous (aNULL)	OPENSSL_EXTRA / OPENSSL_ALL (also accepted as the entire string with no other tokens)
HIGH	All suites except static, anonymous, and NULL ciphers	OPENSSL_EXTRA / OPENSSL_ALL (also accepted as the entire string with no other tokens)
LOW / MEDIUM	Accepted but do not restrict by bit size; behave as “RSA”	OPENSSL_EXTRA / OPENSSL_ALL
aNULL	Anonymous (no authentication) suites	OPENSSL_EXTRA / OPENSSL_ALL; suites require HAVE_ANON
eNULL / NULL	Null encryption suites	OPENSSL_EXTRA / OPENSSL_ALL; suites require HAVE_NULL_CIPHER
kDH	Static DH key exchange	OPENSSL_EXTRA / OPENSSL_ALL
DHE / EDH	Ephemeral DH key exchange	OPENSSL_EXTRA / OPENSSL_ALL; suites require !NO_DH
ECDHE / EECDH	Ephemeral ECDH key exchange	OPENSSL_EXTRA / OPENSSL_ALL; suites require HAVE_ECC
kRSA / RSA	Static RSA key exchange	OPENSSL_EXTRA / OPENSSL_ALL; suites require !NO_RSA
PSK	Pre-shared-key suites	OPENSSL_EXTRA / OPENSSL_ALL; suites require !NO_PSK
DSS	Silently ignored — wolfSSL has no DSA ciphersuites	OPENSSL_EXTRA / OPENSSL_ALL
EXP / EXPORT	Silently ignored — export-grade ciphers are not supported	OPENSSL_EXTRA / OPENSSL_ALL
AES128 / SHA1 / RC4	When negated (“!AES128”, etc.), disable that class	WOLFSSL_SYS_CRYPTO_POLICY
@SECLEVEL=n	Set OpenSSL-compatible security level	WOLFSSL_SYS_CRYPTO_POLICY

Representative TLS 1.3 cipher suite names (each guarded by its own BUILD_* macro; most are enabled

automatically by –enable-tls13):

Name (wolfSSL)	IANA name	Required build option
TLS13-AES128-GCM-SHA256	TLS_AES_128_GCM_SHA256	BUILD_TLS_AES_128_GCM_SHA256 (default with TLS 1.3)
TLS13-AES256-GCM-SHA384	TLS_AES_256_GCM_SHA384	BUILD_TLS_AES_256_GCM_SHA384
TLS13-CHACHA20-POLY1305-SHA256	TLS_CHACHA20_POLY1305_SHA256	BUILD_TLS_CHACHA20_POLY1305_SHA256 (HAVE_CHACHA + HAVE_POLY1305)
TLS13-AES128-CCM-SHA256	TLS_AES_128_CCM_SHA256	BUILD_TLS_AES_128_CCM_SHA256 (HAVE_AESCCM)
TLS13-AES128-CCM-8-SHA256	TLS_AES_128_CCM_8_SHA256	BUILD_TLS_AES_128_CCM_8_SHA256 (HAVE_AESCCM)
TLS13-SM4-GCM-SM3	TLS_SM4_GCM_SM3	BUILD_TLS_SM4_GCM_SM3 (WOLFSSL_SM4_GCM + WOLFSSL_SM3)
TLS13-SM4-CCM-SM3	TLS_SM4_CCM_SM3	BUILD_TLS_SM4_CCM_SM3 (WOLFSSL_SM4_CCM + WOLFSSL_SM3)
TLS13-SHA256-SHA256	TLS_SHA256_SHA256	BUILD_TLS_SHA256_SHA256 (integrity-only)
TLS13-SHA384-SHA384	TLS_SHA384_SHA384	BUILD_TLS_SHA384_SHA384

Representative TLS 1.2 cipher suite name families (each individual suite is guarded by its own BUILD_* macro derived from the IANA name; the common build-option requirements are summarized below):

Name family / example	Typical requirements
ECDHE-ECDSA-AES128-GCM-SHA256, ECDHE-ECDSA-AES256-GCM-SHA384, ECDHE-ECDSA-CHACHA20-POLY1305	HAVE_ECC, HAVE_AESGCM (or HAVE_CHACHA + HAVE_POLY1305)
ECDHE-RSA-AES128-GCM-SHA256, ECDHE-RSA-AES256-GCM-SHA384, ECDHE-RSA-CHACHA20-POLY1305	HAVE_ECC, !NO_RSA, HAVE_AESGCM (or HAVE_CHACHA + HAVE_POLY1305)
DHE-RSA-AES128-GCM-SHA256, DHE-RSA-AES256-GCM-SHA384, DHE-RSA-CHACHA20-POLY1305	!NO_DH, !NO_RSA, HAVE_AESGCM (or HAVE_CHACHA + HAVE_POLY1305)
AES128-SHA, AES256-SHA, AES128-SHA256, AES256-SHA256, AES128-GCM-SHA256, AES256-GCM-SHA384 (static-RSA)	!NO_RSA, !NO_AES_CBC and/or HAVE_AESGCM
DES-CBC3-SHA, RC4-SHA, RC4-MD5	Legacy: !NO_DES3 / !NO_RC4, !NO_OLD_TLS
NULL-SHA, NULL-SHA256, NULL-MD5	HAVE_NULL_CIPHER
PSK-AES128-CBC-SHA256, PSK-AES256-GCM-SHA384, ECDHE-PSK-AES128-CBC-SHA256, DHE-PSK-AES256-GCM-SHA384	!NO_PSK (HAVE_ECC for ECDHE-PSK, !NO_DH for DHE-PSK)
ADH-AES128-SHA, ADH-AES256-SHA	HAVE_ANON

Name family / example	Typical requirements
ECDHE-ECDSA-SM4-GCM-SM3, ECDHE-ECDSA-SM4-CCM-SM3	WOLFSSL_SM2, WOLFSSL_SM3, WOLFSSL_SM4_GCM/WOLFSSL_SM4_CCM

Notes:

- TLS 1.3 suite names and TLS 1.2 suite names may be mixed in the same list; wolfSSL groups them by version internally.
- For DTLS, RC4-based stream ciphers in the list are silently dropped.
- When set as the literal whole-string “DEFAULT”, “ALL”, “HIGH”, or the empty string, wolfSSL installs its built-in default suite list and returns success without parsing further tokens.

Example

```
WOLFSSL_CTX* ctx = 0;
...
ret = wolfSSL_CTX_set_cipher_list(ctx,
    "TLS13-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:"
    "DHE-RSA-AES256-SHA256");
if (ret != WOLFSSL_SUCCESS) {
    // failed to set cipher suite list
}
```

```
int wolfSSL_set_cipher_list(
    WOLFSSL * ssl,
    const char * list
)
```

This function sets the cipher suite list for a given WOLFSSL session. The list format and the set of recognized suite names and keywords are identical to those documented for wolfSSL_CTX_set_cipher_list(); refer to that function for the full keyword/suite tables and required build options. Each call replaces the previous list on the session.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **list** null-terminated text string and a colon-delimited list of cipher suites and/or keywords to use with the specified SSL session.

See:

- wolfSSL_CTX_set_cipher_list
- wolfSSL_get_ciphers
- wolfSSL_get_cipher_list
- wolfSSL_new

Return:

- WOLFSSL_SUCCESS will be returned upon successful function completion.
- WOLFSSL_FAILURE will be returned on failure.

Example

```
int ret = 0;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_set_cipher_list(ssl,
    "TLS13-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:"
    "DHE-RSA-AES256-SHA256");
if (ret != WOLFSSL_SUCCESS) {
    // failed to set cipher suite list
}
```

```
void wolfSSL_dtls13_set_send_more_acks(
    WOLFSSL * ssl,
    int value
)
```

This function sets whether the library should send ACKs to the other peer immediately when detecting disruption or not. Sending ACKs immediately assures minimum latency but it may consume more bandwidth than necessary. If the application manages the timer by itself and this option is set to 0 then application code can use wolfSSL_dtls13_use_quick_timeout() to determine if it should setup a quicker timeout to send those delayed ACKs.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **value** 1 to set the option, 0 to disable the option

See:

- wolfSSL_dtls
- wolfSSL_dtls_get_peer
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls_set_peer
- wolfSSL_dtls13_use_quick_timeout

```
int wolfSSL_dtls_set_timeout_init(
    WOLFSSL * ssl,
    int timeout
)
```

This function sets the dtls timeout.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **timeout** an int type that will be set to the dtls_timeout_init member of the WOLFSSL structure.

See:

- wolfSSL_dtls_set_timeout_max
- wolfSSL_dtls_got_timeout

Return:

- SSL_SUCCESS returned if the function executes without an error. The dtls_timeout_init and the dtls_timeout members of SSL have been set.
- BAD_FUNC_ARG returned if the WOLFSSL struct is NULL or if the timeout is not greater than 0. It will also return if the timeout argument exceeds the maximum value allowed.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
int timeout = TIMEOUT;
...
if(wolfSSL_dtls_set_timeout_init(ssl, timeout)){
    // the dtls timeout was set
} else {
    // Failed to set DTLS timeout.
}
```

```
int wolfSSL_dtls13_do_scheduled_work(
    WOLFSSL * ssl
)
```

Sends the DTLS 1.3 work that was scheduled while reading. With WOLFSSL_RW_THREADED the read path never transmits, because that would race the write thread over the output buffer and the sending key schedule, so ACKs, retransmissions and a KeyUpdate the peer asked for are only sent from the write side. An application that reads without writing must call this, from the same thread it uses for writing, or those messages are never sent and the peer keeps retransmitting what it is waiting to have acknowledged. Not for write-dup applications, which drain through wolfSSL_write().

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_dtls13_pending_work
- wolfSSL_dtls_retransmit

Return:

- WOLFSSL_SUCCESS on success, including when there was nothing to do.
- WOLFSSL_FATAL_ERROR if ssl is NULL, is not a DTLS 1.3 object, is part of a write-dup pair, or the send failed. Call wolfSSL_get_error() to tell a retryable SSL_ERROR_WANT_WRITE from a real failure.

A KeyUpdate is only sent while none of ours is still unacknowledged, since DTLS must not have two in flight. Completing one we started needs the peer's acknowledgement processed, which rotates the sending keys and so does not happen here, so in a WOLFSSL_RW_THREADED build a peer request arriving after that point is held rather than answered.

A send that could only write part of a record returns WOLFSSL_FATAL_ERROR with wolfSSL_get_error() reporting SSL_ERROR_WANT_WRITE. That is not a failure of the connection: the record is held and the next call sends the rest, so wolfSSL_dtls13_pending_work() keeps reporting work until it is out. Treat it as a retry rather than as a reason to stop draining.

Example

```
WOLFSSL* ssl;
...
while (wolfSSL_dtls13_pending_work(ssl)) {
    if (wolfSSL_dtls13_do_scheduled_work(ssl) != WOLFSSL_SUCCESS) {
        if (wolfSSL_get_error(ssl, 0) == SSL_ERROR_WANT_WRITE) {
            // the socket is full, wait for it and call again
            break;
        }
        // a real error
        break;
    }
}
```

```
int wolfSSL_dtls13_pending_work(
    WOLFSSL * ssl
)
```

Reports whether the object has DTLS 1.3 work waiting to be sent by wolfSSL_dtls13_do_scheduled_work(). Only meaningful with WOLFSSL_RW_THREADED. The answer is advisory and can change as soon as it is returned. Only work the pump can actually carry out is reported, so a drain loop over the pair terminates; waiting for the peer to acknowledge a key update we sent is not reported, as there is nothing to send for it. A record the pump could only write in part is reported, so that the retry it owes is not lost.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_dtls13_do_scheduled_work

Return:

- 1 if there is work to send, or if it could not be determined.
- 0 if there is nothing to do, or ssl is not supported here.

Example

```
WOLFSSL* ssl;
...
if (wolfSSL_dtls13_pending_work(ssl))
    wolfSSL_dtls13_do_scheduled_work(ssl);
```

```
WOLFSSL_SESSION * wolfSSL_get1_session(
    WOLFSSL * ssl
)
```

This function returns the WOLFSSL_SESSION from the WOLFSSL structure as a reference type. This requires calling wolfSSL_SESSION_free to release the session reference. The WOLFSSL_SESSION pointed to contains all the necessary information required to perform a session resumption and reestablish the connection without a new handshake. For session resumption, before calling wolfSSL_shutdown() with your session object, an application should save the session ID from the object with a call to wolfSSL_get1_session(), which returns a pointer to the session. Later, the application should create a new WOLFSSL object and assign the saved session with wolfSSL_set_session(). At this point, the application may call wolfSSL_connect() and wolfSSL will try to resume the session. The wolfSSL server code allows session resumption by default. The object returned by wolfSSL_get1_session() needs to be freed after the application is done with it by calling wolfSSL_SESSION_free() on it.

Parameters:

- **ssl** WOLFSSL structure to get session from.

See:

- wolfSSL_new
- wolfSSL_free
- wolfSSL_SESSION_free

Return:

- WOLFSSL_SESSION On success return session pointer.
- NULL will be returned if ssl is NULL, the SSL session cache is disabled, wolfSSL doesn't have the Session ID available, or mutex functions fail.

Example

```
WOLFSSL* ssl;
WOLFSSL_SESSION* ses;
// attempt/complete handshake
wolfSSL_connect(ssl);
ses  = wolfSSL_get1_session(ssl);
// check ses information
// disconnect / setup new SSL instance
wolfSSL_set_session(ssl, ses);
```

```
// attempt/resume handshake
wolfSSL_SESSION_free(ses);
```

```
WOLFSSL_METHOD * wolfSSLv23_client_method(
    void
)
```

The wolfSSLv23_client_method() function is used to indicate that the application is a client and will support the highest protocol version supported by the server between SSL 3.0 - TLS 1.3. This function allocates memory for and initializes a new WOLFSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). Both wolfSSL clients and servers have robust version downgrade capability. If a specific protocol version method is used on either side, then only that version will be negotiated or an error will be returned. For example, a client that uses TLSv1 and tries to connect to a SSLv3 only server will fail, likewise connecting to a TLSv1.1 will fail as well. To resolve this issue, a client that uses the wolfSSLv23_client_method() function will use the highest protocol version supported by the server and downgrade to SSLv3 if needed. In this case, the client will be able to connect to a server running SSLv3 - TLSv1.3.

Parameters:

- **none** No parameters

See:

- wolfSSLv3_client_method
- wolfTLSv1_client_method
- wolfTLSv1_1_client_method
- wolfTLSv1_2_client_method
- wolfTLSv1_3_client_method
- wolfDTLSv1_client_method
- wolfSSL_CTX_new

Return:

- pointer upon success a pointer to a WOLFSSL_METHOD.
- Failure If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;
method = wolfSSLv23_client_method();
if (method == NULL) {
// unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

```
WOLFSSL_BIGNUM * wolfSSL_ASN1_INTEGER_to_BN(
    const WOLFSSL_ASN1_INTEGER * ai,
    WOLFSSL_BIGNUM * bn
)
```

This function is used to copy a WOLFSSL_ASN1_INTEGER value to a WOLFSSL_BIGNUM structure.

Parameters:

- **ai** WOLFSSL_ASN1_INTEGER structure to copy from.
- **bn** if wanting to copy into an already existing WOLFSSL_BIGNUM struct then pass in a pointer to it. Optionally this can be NULL and a new WOLFSSL_BIGNUM structure will be created.

See: none

Return:

- pointer On successfully copying the WOLFSSL_ASN1_INTEGER value a WOLFSSL_BIGNUM pointer is returned.
- Null upon failure.

Example

```
WOLFSSL_ASN1_INTEGER* ai;
WOLFSSL_BIGNUM* bn;
// create ai
bn = wolfSSL_ASN1_INTEGER_to_BN(ai, NULL);

// or if having already created bn and wanting to reuse structure
// wolfSSL_ASN1_INTEGER_to_BN(ai, bn);
// check bn is or return value is not NULL
```

```
long wolfSSL_CTX_add_extra_chain_cert(
    WOLFSSL_CTX * ctx,
    WOLFSSL_X509 * x509
)
```

This function adds the certificate to the internal chain being built in the WOLFSSL_CTX structure.

Parameters:

- **ctx** WOLFSSL_CTX structure to add certificate to.
- **x509** certificate to add to the chain.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free

Return:

- SSL_SUCCESS after successfully adding the certificate.
- SSL_FAILURE if failing to add the certificate to the chain.

Example

```
WOLFSSL_CTX* ctx;
WOLFSSL_X509* x509;
int ret;
// create ctx
ret = wolfSSL_CTX_add_extra_chain_cert(ctx, x509);
// check ret value
```

```
int wolfSSL_CTX_get_read_ahead(
    WOLFSSL_CTX * ctx
)
```

This function returns the get read ahead flag from a WOLFSSL_CTX structure.

Parameters:

- **ctx** WOLFSSL_CTX structure to get read ahead flag from.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free
- wolfSSL_CTX_set_read_ahead

Return:

- flag On success returns the read ahead flag.
- SSL_FAILURE If ctx is NULL then SSL_FAILURE is returned.

Example

```
WOLFSSL_CTX* ctx;
int flag;
// setup ctx
flag = wolfSSL_CTX_get_read_ahead(ctx);
//check flag
```

```
int wolfSSL_CTX_set_read_ahead(
    WOLFSSL_CTX * ctx,
    int v
)
```

This function sets the read ahead flag in the WOLFSSL_CTX structure. When enabled, the record-header read pulls in up to a full TLS record in a single recv(), so the body (and any following buffered records) is obtained without a second syscall. The flag only changes I/O behaviour when the library is built with read-ahead support (–enable-readahead / WOLFSSL_TLS_READ_AHEAD); otherwise it is stored but inert.

Parameters:

- **ctx** WOLFSSL_CTX structure to set read ahead flag.
- **v** read ahead flag

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free
- wolfSSL_CTX_get_read_ahead
- wolfSSL_has_pending

Return:

- SSL_SUCCESS If ctx read ahead flag set.
- SSL_FAILURE If ctx is NULL then SSL_FAILURE is returned.

Note: With read-ahead enabled, undecrypted data can remain buffered internally while the socket has no more data to read. wolfSSL_has_pending() reports whether any such data is buffered, but a non-zero return does not guarantee a full record is available: wolfSSL_read() may still return WANT_READ when only a partial record is buffered. Event_driven applications should therefore drain the connection by calling wolfSSL_read() until it returns WANT_READ (or an error) before returning to select()/poll(), rather than looping on wolfSSL_has_pending() alone; otherwise buffered data could be missed or the loop could spin.

Example

```
WOLFSSL_CTX* ctx;
int flag;
int ret;
// setup ctx
ret = wolfSSL_CTX_set_read_ahead(ctx, flag);
// check return value
```

```
int wolfSSL_CTX_set_default_read_buffer_len(
    WOLFSSL_CTX * ctx,
    size_t len
)
```

This function sets the read-ahead receive window size for contexts created from this WOLFSSL_CTX. When read-ahead is enabled (wolfSSL_CTX_set_read_ahead()), a single recv() pulls in up to `len` bytes:

Parameters:

- **ctx** WOLFSSL_CTX structure to set the read-ahead buffer length on.
- **len** read-ahead coalescing buffer size in bytes (0 = one record).

See:

- wolfSSL_CTX_set_read_ahead
- wolfSSL_set_default_read_buffer_len
- wolfSSL_has_pending

Return:

- SSL_SUCCESS If the buffer length was set.
- SSL_FAILURE If ctx is NULL.

Note: This is a memory-vs-syscall trade-off. While read-ahead is enabled the input buffer is retained (bounded to `len`) for the connection's lifetime, so a large `len` multiplied by many concurrent connections is persistent memory, while a small `len` bounds per-connection footprint at the cost of more syscalls for records that exceed it.

- `len` of 0 resets the window to the one-record default. This is also the window a freshly created context already carries, so the record body is read together with its header without a second syscall.
- A `len` larger than one record lets a single recv() coalesce several back-to-back records, one syscall instead of one per record.
- A `len` smaller than one record caps the receive buffer's footprint when the peer's records are known to be small (e.g. 4 KB), saving heap versus the one-record default. `len` is only a speculative read window, not a hard limit: a record larger than `len` is still received correctly, with the input buffer grown to the record's actual size on demand (costing an extra syscall and reallocation for that record) and then reallocated back down to the window once the oversized record is consumed, so the retained footprint stays bounded by `len` rather than by the largest record seen. The setting only affects I/O when the library is built with read-ahead support (–enable-readahead / WOLFSSL_TLS_READ_AHEAD); otherwise it is stored but inert. This is the wolfSSL equivalent of OpenSSL's SSL_CTX_set_default_read_buffer_len(); unlike OpenSSL it returns a status code (callers that ignore the return remain source-compatible) and it honours sizes below one record, whereas OpenSSL only ever enlarges the buffer. A `len` above WOLFSSL_MAX_READ_AHEAD_SZ (16 MB) is clamped to that maximum. Because 0 and oversized values are both normalised, wolfSSL_CTX_get_default_read_buffer_len() reports the effective window (the one_record default rather than 0 when unset), not the raw argument.

Example

```
WOLFSSL_CTX* ctx;
// setup ctx
wolfSSL_CTX_set_read_ahead(ctx, 1);
// coalesce up to four max-size records per recv()
wolfSSL_CTX_set_default_read_buffer_len(ctx, 4 * 16384);
```

```
int wolfSSL_set_default_read_buffer_len(
    WOLFSSL * ssl,
    size_t len
```

)

This function sets the read-ahead coalescing buffer size on a single WOLFSSL session, overriding the value inherited from its WOLFSSL_CTX. See wolfSSL_CTX_set_default_read_buffer_len() for the full description, the one-record default, and the memory-vs-syscall trade-off.

Parameters:

- **ssl** WOLFSSL structure to set the read-ahead buffer length on.
- **len** read-ahead coalescing buffer size in bytes (0 = one record).

See:

- wolfSSL_CTX_set_default_read_buffer_len
- wolfSSL_set_read_ahead
- wolfSSL_has_pending

Return:

- SSL_SUCCESS If the buffer length was set.
- SSL_FAILURE If ssl is NULL.

```
long wolfSSL_CTX_get_default_read_buffer_len(
    WOLFSSL_CTX * ctx
)
```

This function returns the read-ahead coalescing buffer size configured on a WOLFSSL_CTX by wolfSSL_CTX_set_default_read_buffer_len(). Because a length of 0 and oversized values are normalised when set, the value reported is the effective window actually in use (the one_record default rather than 0 when the length was never changed), not the raw argument last passed.

Parameters:

- **ctx** WOLFSSL_CTX structure to get the read-ahead buffer length from.

See:

- wolfSSL_CTX_set_default_read_buffer_len
- wolfSSL_get_default_read_buffer_len
- wolfSSL_CTX_set_read_ahead

Return:

- len On success returns the read-ahead buffer length in bytes.
- SSL_FAILURE If ctx is NULL.

Example

```
WOLFSSL_CTX* ctx;
long len;
// setup ctx
len = wolfSSL_CTX_get_default_read_buffer_len(ctx);
// check len
```

```
long wolfSSL_get_default_read_buffer_len(
    const WOLFSSL * ssl
)
```

This function returns the read-ahead coalescing buffer size in effect for a single WOLFSSL session, whether inherited from its WOLFSSL_CTX or overridden by wolfSSL_set_default_read_buffer_len(). As with the WOLFSSL_CTX getter, the value reported is the effective window in use, not the raw argument last passed.

Parameters:

- **ssl** WOLFSSL structure to get the read-ahead buffer length from.

See:

- wolfSSL_set_default_read_buffer_len
- wolfSSL_CTX_get_default_read_buffer_len
- wolfSSL_set_read_ahead

Return:

- len On success returns the read-ahead buffer length in bytes.
- SSL_FAILURE If ssl is NULL.

Example

```
WOLFSSL* ssl;
long len;
// setup ssl
len = wolfSSL_get_default_read_buffer_len(ssl);
// check len
```

```
int wolfSSL_CTX_set_max_send_fragment(
    WOLFSSL_CTX * ctx,
    long max_fragment
)
```

This function caps the plaintext carried in each record sent by WOLFSSL objects created from the context afterwards. It is the wolfSSL equivalent of OpenSSL's SSL_CTX_set_max_send_fragment(). The cap is local and send only: the peer is not told and records received are not limited. Unset, there is no cap. wolfSSL_new() copies the value, so a later call does not change existing objects, as in OpenSSL. With a negotiated max_fragment_length the smaller of the two limits applies, where OpenSSL uses

the negotiated length instead. A DTLS write larger than the cap fails with DTLS_SIZE_ERROR, like one larger than the MTU, unless WOLFSSL_NO_DTLS_SIZE_CHECK is defined. In TLS 1.3 only Certificate and CertificateVerify are split to the cap. Other handshake messages, such as ClientHello, ServerHello, EncryptedExtensions, CertificateRequest and NewSessionTicket, are sent in one record each, as they are with a negotiated max_fragment_length. Application data is always split.

Parameters:

- **ctx** WOLFSSL_CTX structure to set the cap on.
- **max_fragment** most plaintext bytes in a record sent.

See:

- wolfSSL_set_max_send_fragment
- wolfSSL_CTX_UseMaxFragment
- wolfSSL_GetMaxOutputSize

Return:

- WOLFSSL_SUCCESS (1) If the cap was set.
- WOLFSSL_FAILURE (0) If ctx is NULL or max_fragment is not in the range 512 to 16384.

Example

```
WOLFSSL_CTX* ctx;
// setup ctx
if (wolfSSL_CTX_set_max_send_fragment(ctx, 1024) != WOLFSSL_SUCCESS) {
    // value out of range
}
```

```
int wolfSSL_set_max_send_fragment(
    WOLFSSL * ssl,
    long max_fragment
)
```

This function caps the plaintext carried in each record sent by a single WOLFSSL object, replacing the value copied from its WOLFSSL_CTX. It is the wolfSSL equivalent of OpenSSL's SSL_set_max_send_fragment() and takes effect from the next record sent. See wolfSSL_CTX_set_max_send_fragment() for the full description.

Parameters:

- **ssl** WOLFSSL structure to set the cap on.
- **max_fragment** most plaintext bytes in a record sent.

See:

- wolfSSL_CTX_set_max_send_fragment
- wolfSSL_UseMaxFragment

- wolfSSL_GetMaxOutputSize

Return:

- WOLFSSL_SUCCESS (1) If the cap was set.
- WOLFSSL_FAILURE (0) If ssl is NULL or max_fragment is not in the range 512 to 16384.

Example

```
WOLFSSL* ssl;
// setup ssl
if (wolfSSL_set_max_send_fragment(ssl, 512) != WOLFSSL_SUCCESS) {
    // value out of range
}
```

```
long wolfSSL_CTX_set_tlsext_opaque_prf_input_callback_arg(
    WOLFSSL_CTX * ctx,
    void * arg
)
```

This function sets the optional argument to be passed to the PRF callback.

Parameters:

- **ctx** WOLFSSL_CTX structure to set user argument.
- **arg** user argument.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free

Return:

- SSL_FAILURE If ctx is NULL.
- SSL_SUCCESS If successfully set.

Example

```
WOLFSSL_CTX* ctx;
void* data;
int ret;
// setup ctx
ret = wolfSSL_CTX_set_tlsext_opaques_prf_input_callback_arg(ctx, data);
//check ret value
```

```
long wolfSSL_CTX_set_options(
    WOLFSSL_CTX * ctx,
    long opt
)
```

This function ORs the bits in opt into the options mask of the given WOLFSSL_CTX. The options mask is inherited by every WOLFSSL session later created from this context. Bits are accumulated — to remove an option, use wolfSSL_CTX_clear_options(). The OpenSSL-style "SSL_OP_" *macros are aliases for the corresponding "WOLFSSL_OP_"* values; either spelling may be used.

Parameters:

- **ctx** WOLFSSL_CTX structure on which to set the options mask.
- **opt** the bitmask of SSL_OP_* / WOLFSSL_OP_* flags to OR into the current mask.

See:

- wolfSSL_CTX_clear_options
- wolfSSL_CTX_get_options
- wolfSSL_set_options
- wolfSSL_get_options

Return:

- BAD_FUNC_ARG if ctx is NULL.
- The updated options mask value stored in ctx on success.

Effective options:

Macro	Effect
SSL_OP_NO_SSLv2	Disable SSLv2 (wolfSSL never supports SSLv2; flag is accepted for OpenSSL compatibility)
SSL_OP_NO_SSLv3	Disable SSLv3
SSL_OP_NO_TLSv1	Disable TLS 1.0
SSL_OP_NO_TLSv1_1	Disable TLS 1.1
SSL_OP_NO_TLSv1_2	Disable TLS 1.2
SSL_OP_NO_TLSv1_3	Disable TLS 1.3 (requires WOLFSSL_TLS13)
SSL_OP_NO_COMPRESSION	Disable record-layer compression (no-op unless HAVE_LIBZ)
SSL_OP_NO_TICKET	Disable RFC 5077 session tickets (TLS 1.2 only; TLS 1.3 ignores this flag); requires HAVE_SESSION_TICKET and (OPENSSL_EXTRA or HAVE_WEBSERVER or WOLFSSL_WPAS_SMALL)
SSL_OP_NO_RENEGOTIATION	Reject peer-initiated renegotiation
SSL_OP_NO_SESSION_RESUMPTION_ON_RENEGOTIATION	Disable session resumption on renegotiation
SSL_OP_COOKIE_EXCHANGE	Enable HelloVerifyRequest cookie exchange (default-on for DTLS)
SSL_OP_NO_QUERY_MTU	DTLS: do not query the path MTU

Macro	Effect
SSL_OP_CIPHER_SERVER_PREFERENCE	Server uses its own cipher preference order rather than the client's
SSL_OP_SINGLE_DH_USE	Generate a fresh DH key for every handshake
SSL_OP_SINGLE_ECDH_USE	Generate a fresh ECDH key for every handshake
SSL_OP_EPHEMERAL_RSA	Use ephemeral RSA (legacy, accepted for OpenSSL compatibility)
SSL_OP_DONT_INSERT_EMPTY_FRAGMENTS	Do not insert empty fragments before a record (CBC BEAST workaround)
SSL_OP_PKCS1_CHECK_1 / _2	Accepted for OpenSSL compatibility
SSL_OP_LEGACY_SERVER_CONNECT	Always allow legacy (unsafe) initial connect; defined as 0 — no effect (requires the openssl/ssl.h compatibility header)

Convenience macros and bug-workaround flags (all members of SSL_OP_ALL, accepted for OpenSSL compatibility but otherwise no-ops in wolfSSL):

- SSL_OP_ALL (bitwise OR of all bug-workaround flags below)
- SSL_OP_MICROSOFT_SESS_ID_BUG
- SSL_OP_NETSCAPE_CHALLENGE_BUG
- SSL_OP_NETSCAPE_REUSE_CIPHER_CHANGE_BUG
- SSL_OP_SSLREF2_REUSE_CERT_TYPE_BUG
- SSL_OP_MICROSOFT_BIG_SSLV3_BUFFER
- SSL_OP_MSIE_SSLV2_RSA_PADDING
- SSL_OP_SSLEAY_080_CLIENT_DH_BUG
- SSL_OP_TLS_D5_BUG
- SSL_OP_TLS_BLOCK_PADDING_BUG
- SSL_OP_TLS_ROLLBACK_BUG
- SSL_OP_NETSCAPE_CA_DN_BUG
- SSL_OP_NETSCAPE_DEMO_CIPHER_CHANGE_BUG

Convenience composite:

- SSL_OP_NO_SSL_MASK = SSL_OP_NO_SSLv3 | SSL_OP_NO_TLSv1 | SSL_OP_NO_TLSv1_1 | SSL_OP_NO_TLSv1_2 | SSL_OP_NO_TLSv1_3

Example

```
WOLFSSL_CTX* ctx;
long mask;
mask = wolfSSL_CTX_set_options(ctx,
    SSL_OP_NO_SSLv3 | SSL_OP_NO_TLSv1 | SSL_OP_NO_TLSv1_1 |
    SSL_OP_NO_COMPRESSION | SSL_OP_CIPHER_SERVER_PREFERENCE);
// 'mask' now reflects the accumulated options stored in ctx
```

```
long wolfSSL_set_options(
    WOLFSSL * s,
    long op
```

)

This function ORs the bits in op into the options mask of the given WOLFSSL session. The set of recognized "SSL_OP_" / "WOLFSSL_OP_" flags is identical to that documented for wolfSSL_CTX_set_options(); refer to that function for the full options table and the build-option requirements of individual flags. Flags inherited from the parent WOLFSSL_CTX remain set; setting SSL_OP_NO_TLSv1_3 here lowers the session's negotiated minor version to TLS 1.2.

Parameters:

- **s** WOLFSSL session on which to set the options mask.
- **op** the bitmask of SSL_OP_* / WOLFSSL_OP_* flags to OR into the current mask.

See:

- wolfSSL_CTX_set_options
- wolfSSL_clear_options
- wolfSSL_get_options
- wolfSSL_new
- wolfSSL_free

Return: The updated options mask value stored in s on success, or 0 if s is NULL.

Example

```
WOLFSSL* ssl;
long mask;
mask = wolfSSL_set_options(ssl, SSL_OP_NO_TLSv1_3);
// 'mask' now reflects the accumulated options stored in ssl
```

```
long wolfSSL_get_options(
    const WOLFSSL * s
)
```

This function returns the current options mask.

Parameters:

- **s** WOLFSSL structure to get options mask from.

See:

- wolfSSL_new
- wolfSSL_free
- wolfSSL_set_options

Return: val Returns the mask value stored in ssl.

Example

```
WOLFSSL* ssl;
unsigned long mask;
mask  = wolfSSL_get_options(ssl);
// check mask
```

```
long wolfSSL_set_tlsext_debug_arg(
    WOLFSSL * s,
    void * arg
)
```

This is used to set the debug argument passed around.

Parameters:

- **s** WOLFSSL structure to set argument in.
- **arg** argument to use.

See:

- wolfSSL_new
- wolfSSL_free

Return:

- SSL_SUCCESS On successful setting argument.
- SSL_FAILURE If an NULL ssl passed in.

Example

```
WOLFSSL* ssl;
void* args;
int ret;
// create ssl object
ret  = wolfSSL_set_tlsext_debug_arg(ssl, args);
// check ret value
```

```
long wolfSSL_set_tlsext_debug_callback(
    WOLFSSL * s,
    WOLFSSL_TLSEXT_DEBUG_CB cb
)
```

This is used to set the TLS extension debug callback on the object.

Parameters:

- **s** WOLFSSL structure to set the callback in.
- **cb** Callback to invoke for each received TLS extension, or NULL to disable it.

See: wolfSSL_set_tlsext_debug_arg

Return:

- WOLFSSL_SUCCESS On successful setting of the callback.
- WOLFSSL_FAILURE If a NULL ssl is passed in.

The callback (type WOLFSSL_TLSEXT_DEBUG_CB) is invoked once for every TLS extension received during the handshake, in wire order, before the extension is processed. It reports the side of the connection, the extension type, the raw extension content and the argument set with wolfSSL_set_tlsext_debug_arg(). Passing a NULL callback disables it.

Example

```
WOLFSSL* ssl;
long ret;
// create ssl object
ret = wolfSSL_set_tlsext_debug_callback(ssl, my_tlsext_debug_cb);
// check ret value
```

```
long wolfSSL_get_verify_result(
    const WOLFSSL * ssl
)
```

This is used to get the results after trying to verify the peer's certificate.

Parameters:

- **ssl** WOLFSSL structure to get verification results from.

See:

- wolfSSL_new
- wolfSSL_free

Return:

- X509_V_OK On successful verification.
- SSL_FAILURE If an NULL ssl passed in.

Example

```
WOLFSSL* ssl;
long ret;
// attempt/complete handshake
ret  = wolfSSL_get_verify_result(ssl);
// check ret value
```

```
int wolfSSL_CTX_allow_anon_cipher(
    WOLFSSL_CTX * ctx
)
```

This function enables the havAnon member of the CTX structure if HAVE_ANON is defined during compilation.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

See: none

Return:

- SSL_SUCCESS returned if the function executed successfully and the haveAnnon member of the CTX is set to 1.
- SSL_FAILURE returned if the CTX structure was NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
#ifdef HAVE_ANON
if(cipherList == NULL){
    wolfSSL_CTX_allow_anon_cipher(ctx);
    if(wolfSSL_CTX_set_cipher_list(ctx, “ADH_AES128_SHA”) != SSL_SUCCESS){
        // failure case
    }
}
#endif
```

```
WOLFSSL_METHOD * wolfSSLv23_server_method(
    void
)
```

The wolfSSLv23_server_method() function is used to indicate that the application is a server and will support clients connecting with protocol version from SSL 3.0 - TLS 1.3. This function allocates memory for and initializes a new WOLFSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Parameters:

- **none** No parameters

See:

- wolfSSLv3_server_method
- wolfTLSv1_server_method

- wolfTLSv1_1_server_method
- wolfTLSv1_2_server_method
- wolfTLSv1_3_server_method
- wolfDTLSv1_server_method
- wolfSSL_CTX_new

Return:

- pointer If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- Failure If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfSSLv23_server_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

```
int wolfSSL_state(
    WOLFSSL * ssl
)
```

This is used to get the internal error state of the WOLFSSL structure.

See:

- wolfSSL_new
- wolfSSL_free

Return:

- wolfssl_error returns ssl error state, usually a negative
- BAD_FUNC_ARG if ssl is NULL.
- ssl WOLFSSL structure to get state from.

Example

```
WOLFSSL* ssl;
int ret;
// create ssl object
ret  = wolfSSL_state(ssl);
// check ret value
```

```
int wolfSSL_check_domain_name(
    WOLFSSL * ssl,
    const char * dn
)
```

wolfSSL by default checks the peer certificate for a valid date range and a verified signature. Calling this function before wolfSSL_connect() or wolfSSL_accept() will add a domain name check to the list of checks to perform. dn holds the domain name to check against the peer certificate when it's received.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **dn** domain name to check against the peer certificate when received.

See: none

Return:

- SSL_SUCCESS upon success.
- SSL_FAILURE will be returned if a memory error was encountered.

Example

```
int ret = 0;
WOLFSSL* ssl;
char* domain = (char*) "www.yassl.com";
...

ret = wolfSSL_check_domain_name(ssl, domain);
if (ret != SSL_SUCCESS) {
   // failed to enable domain name check
}
```

```
int wolfSSL_check_ip_address(
    WOLFSSL * ssl,
    const char * ipaddr
)
```

Calling this function before wolfSSL_connect() or wolfSSL_accept() adds an IP-address identity check against the peer certificate SAN iPAddress entries.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **ipaddr** NULL-terminated ASCII IP address string to verify against the peer certificate.

See: wolfSSL_check_domain_name

Return:

- SSL_SUCCESS upon success.
- SSL_FAILURE if parameters are invalid or memory allocation fails.

Example

```
int ret = 0;
WOLFSSL* ssl;
const char* ip = "127.0.0.1";
...

ret = wolfSSL_check_ip_address(ssl, ip);
if (ret != SSL_SUCCESS) {
   // failed to enable IP check
}
```

```
int wolfSSL_set_compression(
    WOLFSSL * ssl
)
```

Turns on the ability to use compression for the SSL connection. Both sides must have compression turned on otherwise compression will not be used. The zlib library performs the actual data compression. To compile into the library use –with-libz for the configure system and define HAVE_LIBZ otherwise. Keep in mind that while compressing data before sending decreases the actual size of the messages being sent and received, the amount of data saved by compression usually takes longer in time to analyze than it does to send it raw on all but the slowest of networks.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See: none

Return:

- SSL_SUCCESS upon success.
- NOT_COMPILED_IN will be returned if compression support wasn't built into the library.
- BAD_FUNC_ARG will be returned if ssl is NULL or is a DTLS session.

Record layer compression was removed in TLS 1.3 (RFC 8446 section 5.2). A connection that negotiates TLS 1.3 or DTLS 1.3 therefore completes uncompressed and this request is silently dropped; the call still returns SSL_SUCCESS, so check the negotiated protocol version rather than this return value to learn whether compression is actually in use. For the same reason a ClientHello that offers TLS 1.3 never advertises zlib.

Compression is not available over DTLS at all. zlib keeps one deflate stream running across records, so a datagram that is lost, duplicated or reordered would desync the peer for the rest of the connection. Since the transport is known when the WOLFSSL object is created, this is reported rather than dropped.

Example

```
int ret = 0;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_set_compression(ssl);
if (ret == SSL_SUCCESS) {
    // successfully enabled compression for SSL session
}
```

```
int wolfSSL_set_timeout(
    WOLFSSL * ssl,
    unsigned int to
)
```

This function sets the SSL session timeout value in seconds.

Parameters:

- **ssl** pointer to the SSL object, created with wolfSSL_new().
- **to** value, in seconds, used to set the SSL session timeout.

See:

- wolfSSL_get1_session
- wolfSSL_set_session

Return:

- SSL_SUCCESS will be returned upon successfully setting the session.
- BAD_FUNC_ARG will be returned if ssl is NULL.

Example

```
int ret = 0;
WOLFSSL* ssl = 0;
...

ret = wolfSSL_set_timeout(ssl, 500);
if (ret != SSL_SUCCESS) {
    // failed to set session timeout value
}
...
```

```
int wolfSSL_CTX_set_timeout(
    WOLFSSL_CTX * ctx,
    unsigned int to
)
```

This function sets the timeout value for SSL sessions, in seconds, for the specified SSL context.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **to** session timeout value in seconds.

See:

- wolfSSL_flush_sessions
- wolfSSL_get1_session
- wolfSSL_set_session
- wolfSSL_get_sessionID
- wolfSSL_CTX_set_session_cache_mode

Return:

- the previous timeout value, if WOLFSSL_ERROR_CODE_OPENSSL is
- defined on success. If not defined, SSL_SUCCESS will be returned.
- BAD_FUNC_ARG will be returned when the input context (ctx) is null.

Example

```
WOLFSSL_CTX*    ctx    = 0;
...
ret = wolfSSL_CTX_set_timeout(ctx, 500);
if (ret != SSL_SUCCESS) {
    // failed to set session timeout value
}
```

```
int wolfSSL_CTX_UnloadCAs(
    WOLFSSL_CTX * ctx
)
```

This function unloads the CA signer list and frees the whole signer table.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

See:

- wolfSSL_CertManagerUnloadCAs
- LockMutex
- UnlockMutex

Return:

- SSL_SUCCESS returned on successful execution of the function.

- BAD_FUNC_ARG returned if the WOLFSSL_CTX struct is NULL or there are otherwise unpermitted argument values passed in a subroutine.
- BAD_MUTEX_E returned if there was a mutex error. The LockMutex() did not return 0.

Example

```
WOLFSSL_METHOD method = wolfTLSv1_2_client_method();
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(method);
…
if(wolfSSL_CTX_UnloadCAs(ctx) != SSL_SUCCESS){
    // The function did not unload CAs
}
```

```
int wolfSSL_CTX_UnloadIntermediateCerts(
    WOLFSSL_CTX * ctx
)
```

This function unloads intermediate certificates added to the CA signer list and frees them.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

See:

- wolfSSL_CTX_UnloadCAs
- wolfSSL_CertManagerUnloadIntermediateCerts

Return:

- SSL_SUCCESS returned on successful execution of the function.
- BAD_FUNC_ARG returned if the WOLFSSL_CTX struct is NULL or there are otherwise unpermitted argument values passed in a subroutine.
- BAD_STATE_E returned if the WOLFSSL_CTX has a reference count > 1.
- BAD_MUTEX_E returned if there was a mutex error. The LockMutex() did not return 0.

Example

```
WOLFSSL_METHOD method = wolfTLSv1_2_client_method();
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(method);
…
if(wolfSSL_CTX_UnloadIntermediateCerts(ctx) != NULL){
    // The function did not unload CAs
}
```

```
int wolfSSL_CTX_Unload_trust_peers(
    WOLFSSL_CTX * ctx
)
```

This function is used to unload all previously loaded trusted peer certificates. Feature is enabled by defining the macro WOLFSSL_TRUST_PEER_CERT.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().

See:

- wolfSSL_CTX_trust_peer_buffer
- wolfSSL_CTX_trust_peer_cert

Return:

- SSL_SUCCESS upon success.
- BAD_FUNC_ARG will be returned if ctx is NULL.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- BAD_MUTEX_E will be returned when locking fails.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_Unload_trust_peers(ctx);
if (ret != SSL_SUCCESS) {
    // error unloading trusted peer certs
}
...
```

```
int wolfSSL_CTX_trust_peer_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format
)
```

This function loads a certificate to use for verifying a peer when performing a TLS/SSL handshake. The peer certificate sent during the handshake is compared by using the SKID when available and the signature. If these two things do not match then any loaded CAs are used. Is the same functionality as wolfSSL_CTX_trust_peer_cert except is from a buffer instead of a file. Feature is enabled by defining the macro WOLFSSL_TRUST_PEER_CERT Please see the examples for proper usage.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **buffer** pointer to the buffer containing certificates.
- **sz** length of the buffer input.
- **type** type of certificate being loaded i.e. SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_use_certificate_file
- wolfSSL_CTX_use_PrivateKey_file
- wolfSSL_CTX_use_certificate_chain_file
- wolfSSL_CTX_trust_peer_cert
- wolfSSL_CTX_Unload_trust_peers
- wolfSSL_use_certificate_file
- wolfSSL_use_PrivateKey_file
- wolfSSL_use_certificate_chain_file

Return:

- SSL_SUCCESS upon success
- SSL_FAILURE will be returned if ctx is NULL, or if both file and type are invalid.
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...

ret = wolfSSL_CTX_trust_peer_buffer(ctx, bufferPtr, bufferSz,
SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
// error loading trusted peer cert
}
...
```

```
int wolfSSL_CTX_set_group_messages(
    WOLFSSL_CTX * ctx
)
```

This function turns on grouping of handshake messages where possible.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().

See:

- wolfSSL_set_group_messages
- wolfSSL_CTX_clear_group_messages
- wolfSSL_clear_group_messages
- wolfSSL_CTX_new

Return:

- SSL_SUCCESS will be returned upon success.
- BAD_FUNC_ARG will be returned if the input context is null.

Example

```
WOLFSSL_CTX* ctx = 0;
...
ret = wolfSSL_CTX_set_group_messages(ctx);
if (ret != SSL_SUCCESS) {
    // failed to set handshake message grouping
}
```

```
int wolfSSL_set_group_messages(
    WOLFSSL * ssl
)
```

This function turns on grouping of handshake messages where possible.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See:

- wolfSSL_CTX_set_group_messages
- wolfSSL_clear_group_messages
- wolfSSL_CTX_clear_group_messages
- wolfSSL_new

Return:

- SSL_SUCCESS will be returned upon success.
- BAD_FUNC_ARG will be returned if the input context is null.

Example

```
WOLFSSL* ssl = 0;
...
ret = wolfSSL_set_group_messages(ssl);
if (ret != SSL_SUCCESS) {
```

```
// failed to set handshake message grouping
}
```

```
int wolfSSL_CTX_SetMinVersion(
    WOLFSSL_CTX * ctx,
    int version
)
```

This function sets the minimum downgrade version allowed. Applicable only when the connection allows downgrade using (wolfSSLv23_client_method or wolfSSLv23_server_method).

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **version** an integer representation of the version to be set as the minimum: WOLFSSL_SSLV3 = 0, WOLFSSL_TLSV1 = 1, WOLFSSL_TLSV1_1 = 2 or WOLFSSL_TLSV1_2 = 3.

See: SetMinVersionHelper

Return:

- SSL_SUCCESS returned if the function returned without error and the minimum version is set.
- BAD_FUNC_ARG returned if the WOLFSSL_CTX structure was NULL or if the minimum version is not supported.

Example

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new( protocol method );
WOLFSSL* ssl = WOLFSSL_new(ctx);
int version; // macrop representation
…
if(wolfSSL_CTX_SetMinVersion(ssl->ctx, version) != SSL_SUCCESS){
    // Failed to set min version
}
```

```
int wolfSSL_SetVersion(
    WOLFSSL * ssl,
    int version
)
```

This function sets the SSL/TLS protocol version for the specified SSL session (WOLFSSL object) using the version as specified by version. This will override the protocol setting for the SSL session (ssl) - originally defined and set by the SSL context (wolfSSL_CTX_new()) method type.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **version** SSL/TLS protocol version. Possible values include WOLFSSL_SSLV3, WOLFSSL_TLSV1, WOLFSSL_TLSV1_1, WOLFSSL_TLSV1_2.

See: wolfSSL_CTX_new

Return:

- SSL_SUCCESS upon success.
- BAD_FUNC_ARG will be returned if the input SSL object is NULL or an incorrect protocol version is given for version.

Example

```
int ret = 0;
WOLFSSL* ssl;
...

ret = wolfSSL_SetVersion(ssl, WOLFSSL_TLSV1);
if (ret != SSL_SUCCESS) {
    // failed to set SSL session protocol version
}
```

```
int wolfSSL_UseALPN(
    WOLFSSL * ssl,
    char * protocol_name_list,
    unsigned int protocol_name_listSz,
    unsigned char options
)
```

Setup ALPN use for a wolfSSL session.

Parameters:

- **ssl** The wolfSSL session to use.
- **protocol_name_list** List of protocol names to use. Comma delimited string is required.
- **protocol_name_listSz** Size of the list of protocol names.
- **options** WOLFSSL_ALPN_CONTINUE_ON_MISMATCH or WOLFSSL_ALPN_FAILED_ON_MISMATCH.

See: TLSX_UseALPN

Return:

- WOLFSSL_SUCCESS: upon success.
- BAD_FUNC_ARG Returned if ssl or protocol_name_list is null or protocol_name_listSz is too large or options contain something not supported.
- MEMORY_ERROR Error allocating memory for protocol list.
- SSL_FAILURE upon failure.

Example

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
```

```
WOLFSSL_METHOD method = // Some wolfSSL method
ctx = wolfSSL_CTX_new(method);
ssl = wolfSSL_new(ctx);

char alpn_list[] = {};

if (wolfSSL_UseALPN(ssl, alpn_list, sizeof(alpn_list),
    WOLFSSL_APN_FAILED_ON_MISMATCH) != WOLFSSL_SUCCESS)
{
   // Error setting session ticket
}
```

```
int wolfSSL_CTX_UseSessionTicket(
    WOLFSSL_CTX * ctx
)
```

This function sets wolfSSL context to use a session ticket.

Parameters:

- **ctx** The WOLFSSL_CTX structure to use.

See: TLSX_UseSessionTicket

Return:

- SSL_SUCCESS Function executed successfully.
- BAD_FUNC_ARG Returned if ctx is null.
- MEMORY_E Error allocating memory in internal function.

Example

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL_METHOD method = // Some wolfSSL method ;
ctx = wolfSSL_CTX_new(method);

if(wolfSSL_CTX_UseSessionTicket(ctx) != SSL_SUCCESS)
{
    // Error setting session ticket
}
```

```
int wolfSSL_check_private_key(
    const WOLFSSL * ssl
)
```

This function checks that the private key is a match with the certificate being used.

Parameters:

- **ssl** WOLFSSL structure to check.

See:

- wolfSSL_new
- wolfSSL_free

Return:

- SSL_SUCCESS On successfully match.
- SSL_FAILURE If an error case was encountered.
- <0 All error cases other than SSL_FAILURE are negative values.

Example

```
WOLFSSL* ssl;
int ret;
// create and set up ssl
ret  = wolfSSL_check_private_key(ssl);
// check ret value
```

```
int wolfSSL_use_certificate(
    WOLFSSL * ssl,
    WOLFSSL_X509 * x509
)
```

his is used to set the certificate for WOLFSSL structure to use during a handshake.

Parameters:

- **ssl** WOLFSSL structure to set certificate in.
- **x509** certificate to use.

See:

- wolfSSL_new
- wolfSSL_free

Return:

- SSL_SUCCESS On successful setting argument.
- SSL_FAILURE If a NULL argument passed in.

Example

```
 WOLFSSL* ssl;
WOLFSSL_X509* x509
int ret;
// create ssl object and x509
```

```
ret  = wolfSSL_use_certificate(ssl, x509);
// check ret value
```

```
int wolfSSL_use_certificate_ASN1(
    WOLFSSL * ssl,
    const unsigned char * der,
    int derSz
)
```

This is used to set the certificate for WOLFSSL structure to use during a handshake. A DER formatted buffer is expected.

Parameters:

- **ssl** WOLFSSL structure to set certificate in.
- **der** DER certificate to use.
- **derSz** size of the DER buffer passed in.

See:

- wolfSSL_new
- wolfSSL_free

Return:

- SSL_SUCCESS On successful setting argument.
- SSL_FAILURE If a NULL argument passed in.

Example

```
WOLFSSL* ssl;
unsigned char* der;
int derSz;
int ret;
// create ssl object and set DER variables
ret  = wolfSSL_use_certificate_ASN1(ssl, der, derSz);
// check ret value
```

```
int wolfSSL_SESSION_get_master_key(
    const WOLFSSL_SESSION * ses,
    unsigned char * out,
    int outSz
)
```

This is used to get the master key after completing a handshake.

Parameters:

- **ses** WOLFSSL_SESSION structure to get master secret buffer from.
- **out** buffer to hold data.
- **outSz** size of out buffer passed in. (if 0 function will return max buffer size needed)

See:

- wolfSSL_new
- wolfSSL_free

Return:

-
  0 On successfully getting data returns a value greater than 0
- 0 If no random data buffer or an error state returns 0
- max If outSz passed in is 0 then the maximum buffer size needed is returned

Example

```
WOLFSSL_SESSION ssl;
unsigned char* buffer;
size_t bufferSz;
size_t ret;
// complete handshake and get session structure
bufferSz  = wolfSSL_SESSION_get_master_secret(ses, NULL, 0);
buffer = malloc(bufferSz);
ret  = wolfSSL_SESSION_get_master_secret(ses, buffer, bufferSz);
// check ret value
```

```
int wolfSSL_SESSION_get_master_key_length(
    const WOLFSSL_SESSION * ses
)
```

This is used to get the master secret key length.

Parameters:

- **ses** WOLFSSL_SESSION structure to get master secret buffer from.

See:

- wolfSSL_new
- wolfSSL_free

Return: size Returns master secret key size.

Example

```
WOLFSSL_SESSION ssl;
unsigned char* buffer;
size_t bufferSz;
size_t ret;
// complete handshake and get session structure
bufferSz  = wolfSSL_SESSION_get_master_secret_length(ses);
buffer = malloc(bufferSz);
// check ret value
```

```
void wolfSSL_CTX_set_cert_store(
    WOLFSSL_CTX * ctx,
    WOLFSSL_X509_STORE * str
)
```

This is a setter function for the WOLFSSL_X509_STORE structure in ctx.

Parameters:

- **ctx** pointer to the WOLFSSL_CTX structure for setting cert store pointer.
- **str** pointer to the WOLFSSL_X509_STORE to set in ctx.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free

Return: none No return.

Example

```
WOLFSSL_CTX ctx;
WOLFSSL_X509_STORE* st;
// setup ctx and st
st = wolfSSL_CTX_set_cert_store(ctx, st);
//use st
```

```
WOLFSSL_X509_STORE * wolfSSL_CTX_get_cert_store(
    WOLFSSL_CTX * ctx
)
```

This is a getter function for the WOLFSSL_X509_STORE structure in ctx.

Parameters:

- **ctx** pointer to the WOLFSSL_CTX structure for getting cert store pointer.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free
- wolfSSL_CTX_set_cert_store

Return:

- WOLFSSL_X509_STORE* On successfully getting the pointer.
- NULL Returned if NULL arguments are passed in.

Example

```
WOLFSSL_CTX ctx;
WOLFSSL_X509_STORE* st;
// setup ctx
st = wolfSSL_CTX_get_cert_store(ctx);
//use st
```

```
size_t wolfSSL_get_server_random(
    const WOLFSSL * ssl,
    unsigned char * out,
    size_t outlen
)
```

This is used to get the random data sent by the server during the handshake.

Parameters:

- **ssl** WOLFSSL structure to get clients random data buffer from.
- **out** buffer to hold random data.
- **outSz** size of out buffer passed in. (if 0 function will return max buffer size needed)

See:

- wolfSSL_new
- wolfSSL_free

Return:

- 0 On successfully getting data returns a value greater than 0
- 0 If no random data buffer or an error state returns 0
- max If outSz passed in is 0 then the maximum buffer size needed is returned

Example

```
WOLFSSL ssl;
unsigned char* buffer;
size_t bufferSz;
```

```
size_t ret;
bufferSz  = wolfSSL_get_server_random(ssl, NULL, 0);
buffer = malloc(bufferSz);
ret  = wolfSSL_get_server_random(ssl, buffer, bufferSz);
// check ret value
```

```
size_t wolfSSL_get_client_random(
    const WOLFSSL * ssl,
    unsigned char * out,
    size_t outSz
)
```

This is used to get the random data sent by the client during the handshake.

Parameters:

- **ssl** WOLFSSL structure to get clients random data buffer from.
- **out** buffer to hold random data.
- **outSz** size of out buffer passed in. (if 0 function will return max buffer size needed)

See:

- wolfSSL_new
- wolfSSL_free

Return:

- 0 On successfully getting data returns a value greater than 0
- 0 If no random data buffer or an error state returns 0
- max If outSz passed in is 0 then the maximum buffer size needed is returned

Example

```
WOLFSSL ssl;
unsigned char* buffer;
size_t bufferSz;
size_t ret;
bufferSz  = wolfSSL_get_client_random(ssl, NULL, 0);
buffer = malloc(bufferSz);
ret  = wolfSSL_get_client_random(ssl, buffer, bufferSz);
// check ret value
```

```
wc_pem_password_cb * wolfSSL_CTX_get_default_passwd_cb(
    WOLFSSL_CTX * ctx
)
```

This is a getter function for the password callback set in ctx.

Parameters:

- **ctx** WOLFSSL_CTX structure to get call back from.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free

Return:

- func On success returns the callback function.
- NULL If ctx is NULL then NULL is returned.

Example

```
WOLFSSL_CTX* ctx;
wc_pem_password_cb cb;
// setup ctx
cb = wolfSSL_CTX_get_default_passwd_cb(ctx);
//use cb
```

```
void * wolfSSL_CTX_get_default_passwd_cb_userdata(
    WOLFSSL_CTX * ctx
)
```

This is a getter function for the password callback user data set in ctx.

Parameters:

- **ctx** WOLFSSL_CTX structure to get user data from.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free

Return:

- pointer On success returns the user data pointer.
- NULL If ctx is NULL then NULL is returned.

Example

```
WOLFSSL_CTX* ctx;
void* data;
// setup ctx
data = wolfSSL_CTX_get_default_passwd_cb(ctx);
//use data
```

```
long wolfSSL_CTX_clear_options(
    WOLFSSL_CTX * ctx,
    long opt
)
```

This function resets option bits of WOLFSSL_CTX object.

Parameters:

- **ctx** pointer to the SSL context.

See:

- wolfSSL_CTX_new
- wolfSSL_new
- wolfSSL_free

Return: option new option bits

Example

```
WOLFSSL_CTX* ctx = 0;
...
wolfSSL_CTX_clear_options(ctx, SSL_OP_NO_TLSv1);
```

```
int wolfSSL_set_msg_callback(
    WOLFSSL * ssl,
    SSL_Msg_Cb cb
)
```

This function sets a callback in the ssl. The callback is to observe handshake messages. NULL value of cb resets the callback.

Parameters:

- **ssl** WOLFSSL structure to set callback argument.

See: wolfSSL_set_msg_callback_arg

Return:

- SSL_SUCCESS On success.
- SSL_FAILURE If an NULL ssl passed in.

Example

```
static cb(int write_p, int version, int content_type,
const void *buf, size_t len, WOLFSSL *ssl, void *arg)
…
WOLFSSL* ssl;
ret  = wolfSSL_set_msg_callback(ssl, cb);
// check ret
```

```
int wolfSSL_set_msg_callback_arg(
    WOLFSSL * ssl,
    void * arg
)
```

This function sets associated callback context value in the ssl. The value is handed over to the callback argument.

Parameters:

- **ssl** WOLFSSL structure to set callback argument.

See: wolfSSL_set_msg_callback

Return: none No return.

Example

```
static cb(int write_p, int version, int content_type,
const void *buf, size_t len, WOLFSSL *ssl, void *arg)
…
WOLFSSL* ssl;
ret  = wolfSSL_set_msg_callback(ssl, cb);
// check ret
wolfSSL_set_msg_callback(ssl, arg);
```

```
int wolfSSL_send_hrr_cookie(
    WOLFSSL * ssl,
    const unsigned char * secret,
    unsigned int secretSz
)
```

This function is called on the server side to indicate that a HelloRetryRequest message must contain a Cookie and, in case of using protocol DTLS v1.3, that the handshake will always include a cookie exchange. Please note that when using protocol DTLS v1.3, the cookie exchange is enabled by default. The Cookie holds a hash of the current transcript so that another server process can handle the ClientHello in reply. The secret is used when generating the integrity check on the Cookie data. This replaces or regenerates the HRR secret, then delegates to wolfSSL_enable_cookie(). Changing cookie mode after handshake processing starts is unsupported; rotating a secret while cookies are already enabled remains supported. DTLS 1.2_only callers should use wolfSSL_enable_cookie() and wolfSSL_DTLS_SetCookieSecret().

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **secret** a pointer to a buffer holding the secret. Passing NULL indicates to generate a new random secret.
- **secretSz** Size of the secret in bytes. Passing 0 indicates to use the default size: the digest size of the hash the cookie MAC uses, WC_SHA256_DIGEST_SIZE, or when SHA-256 is not available WC_SHA384_DIGEST_SIZE, WC_SHA512_DIGEST_SIZE or WC_SM3_DIGEST_SIZE, in that order.

See:

- wolfSSL_new
- wolfSSL_disable_hrr_cookie

Return:

- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3.
- SIDE_ERROR if called with a client.
- BAD_STATE_E if the DTLS handshake already committed to stateful processing, as for wolfSSL_enable_cookie(), or if a secret must be generated (secret is NULL, or a DTLS 1.3 server is missing its DTLS 1.2 secret) and the session's RNG has been released. The installed secret is left unchanged.
- WOLFSSL_SUCCESS if successful.
- MEMORY_ERROR if allocating dynamic memory for storing secret failed.
- Another -ve value on internal error. Every failure is all or nothing: the secret in use and the cookie policy are left exactly as they were found, so the object stays usable.

Example

```
int ret;
WOLFSSL* ssl;
char secret[32];
...
ret = wolfSSL__send_hrr_cookie(ssl, secret, sizeof(secret));
if (ret != WOLFSSL_SUCCESS) {
    // failed to set use of Cookie and secret
}
```

```
int wolfSSL_set_hrr_cookie_secret_secondary(
    WOLFSSL * ssl,
    const unsigned char * secret,
    unsigned int secretSz
)
```

This function sets a secondary HelloRetryRequest cookie secret on a DTLS 1.3 server. It is used only when verifying a received cookie, and only if the primary secret (set with wolfSSL_send_hrr_cookie()) fails to verify it. This lets an application rotate the cookie secret on a stateless DTLS 1.3 server without rejecting clients whose cookie was issued under the previous secret: install the new secret as the primary and the previous secret here for an overlap window. The secondary secret is never used to

issue cookies. This API is DTLS only; TLS 1.3 over a reliable transport does not verify cookies statelessly across the HelloRetryRequest exchange.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **secret** a pointer to a buffer holding the secondary secret. Passing NULL (or a secretSz of 0) clears any previously set secondary secret.
- **secretSz** Size of the secret in bytes.

See:

- wolfSSL_send_hrr_cookie
- wolfSSL_disable_hrr_cookie

Return:

- BAD_FUNC_ARG if ssl is NULL, not using TLS v1.3, or not using DTLS.
- SIDE_ERROR if called with a client.
- WOLFSSL_SUCCESS if successful.
- MEMORY_ERROR if allocating dynamic memory for storing secret failed. The secondary secret already in use, if any, is kept.

Example

```
int ret;
WOLFSSL* ssl;
char newSecret[32];
char oldSecret[32];
...
// rotate: new secret becomes primary, previous secret stays valid
wolfSSL_send_hrr_cookie(ssl, newSecret, sizeof(newSecret));
ret = wolfSSL_set_hrr_cookie_secret_secondary(ssl, oldSecret,
    sizeof(oldSecret));
if (ret != WOLFSSL_SUCCESS) {
    // failed to set the secondary cookie secret
}
```

```
int wolfSSL_disable_hrr_cookie(
    WOLFSSL * ssl
)
```

This function is called on the server side to indicate that a HelloRetryRequest message must NOT contain a Cookie and that, if using protocol DTLS v1.3, a cookie exchange will not be included in the handshake. Please note that not doing a cookie exchange when using protocol DTLS v1.3 can make the server susceptible to DoS/Amplification attacks. This delegates to wolfSSL_disable_cookie(): on a DTLS object it also disables the DTLS 1.2 HelloVerifyRequest exchange, and it fails with BAD_STATE_E once the handshake has committed to stateful processing.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_disable_cookie
- wolfSSL_send_hrr_cookie

Return:

- WOLFSSL_SUCCESS if successful
- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3
- SIDE_ERROR if invoked on a client
- BAD_STATE_E if the DTLS handshake already committed to stateful processing

```
int wolfSSL_CTX_no_ticket_TLSv13(
    WOLFSSL_CTX * ctx
)
```

This function is called on the server to stop it from sending a resumption session ticket once the handshake is complete.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().

See: wolfSSL_no_ticket_TLSv13

Return:

- BAD_FUNC_ARG if ctx is NULL or not using TLS v1.3.
- SIDE_ERROR if called with a client.
- 0 if successful.

Example

```
int ret;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_no_ticket_TLSv13(ctx);
if (ret != 0) {
    // failed to set no ticket
}
```

```
int wolfSSL_no_ticket_TLSv13(
    WOLFSSL * ssl
)
```

This function is called on the server to stop it from sending a resumption session ticket once the handshake is complete.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_CTX_no_ticket_TLSv13

Return:

- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3.
- SIDE_ERROR if called with a client.
- 0 if successful.

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_no_ticket_TLSv13(ssl);
if (ret != 0) {
    // failed to set no ticket
}
```

```
int wolfSSL_CTX_no_dhe_psk(
    WOLFSSL_CTX * ctx
)
```

This function is called on a TLS v1.3 wolfSSL context to disallow Diffie-Hellman (DH) style key exchanges when handshakes are using pre-shared keys for authentication.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().

See: wolfSSL_no_dhe_psk

Return:

- BAD_FUNC_ARG if ctx is NULL or not using TLS v1.3.
- 0 if successful.

Example

```
int ret;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_no_dhe_psk(ctx);
if (ret != 0) {
```

```
    // failed to set no DHE for PSK handshakes
}



int wolfSSL_no_dhe_psk(
    WOLFSSL * ssl
)
```

This function is called on a TLS v1.3 client or server wolfSSL to disallow Diffie-Hellman (DH) style key exchanges when handshakes are using pre-shared keys for authentication.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_CTX_no_dhe_psk

Return:

- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3.
- 0 if successful.

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_no_dhe_psk(ssl);
if (ret != 0) {
    // failed to set no DHE for PSK handshakes
}



int wolfSSL_CTX_require_psk(
    WOLFSSL_CTX * ctx
)
```

This function is called on a TLS v1.3 / DTLS v1.3 context to require that an external Pre-Shared Key is negotiated for the handshake to succeed. When set, a handshake that completes without negotiating an external PSK is aborted with PSK_MISSING_ERROR instead of falling back to a certificate handshake, so the PSK acts as an additional security factor. The requirement keys off the external-PSK callback (it has no effect unless one is registered) and session-ticket resumption is exempt. To preserve forward secrecy a mandatory external PSK must also use an (EC)DHE key exchange; a pure psk_ke handshake is rejected with PSK_KEY_ERROR. This applies to TLS 1.3 and DTLS 1.3 only; in (D)TLS 1.2 the use of a PSK is determined by the negotiated cipher suite, so a mandatory PSK is instead configured by restricting the cipher suite list to (preferably (EC)DHE-)PSK suites.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

See: wolfSSL_require_psk

Return:

- BAD_FUNC_ARG if ctx is NULL or not at least TLS v1.3.
- 0 if successful.

Note: In builds compiled without external-PSK and without session-ticket support (NO_PSK defined and HAVE_SESSION_TICKET undefined) the requirement cannot be enforced; the function still returns 0 but has no effect.

Warning: Because the requirement can only be enforced for (D)TLS 1.3, this function also disables version downgrade on the context so it cannot silently fall back to (D)TLS 1.2 and complete a handshake without a PSK. A peer that does not support (D)TLS 1.3 will therefore fail to connect.

Example

```
int ret;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_require_psk(ctx);
if (ret != 0) {
    // failed to make a PSK mandatory
}
```

```
int wolfSSL_require_psk(
    WOLFSSL * ssl
)
```

This function is called on a TLS v1.3 / DTLS v1.3 wolfSSL object to require that an external Pre-Shared Key is negotiated for the handshake to succeed. See wolfSSL_CTX_require_psk() for the full behaviour.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_CTX_require_psk

Return:

- BAD_FUNC_ARG if ssl is NULL or not at least TLS v1.3.
- 0 if successful.

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_require_psk(ssl);
if (ret != 0) {
```

```
    // failed to make a PSK mandatory
}
```

```
int wolfSSL_CTX_allow_post_handshake_auth(
    WOLFSSL_CTX * ctx
)
```

This function is called on a TLS v1.3 client wolfSSL context to allow a client certificate to be sent post handshake upon request from server. This is useful when connecting to a web server that has some pages that require client authentication and others that don't.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().

See:

- wolfSSL_allow_post_handshake_auth
- wolfSSL_request_certificate

Return:

- BAD_FUNC_ARG if ctx is NULL or not using TLS v1.3.
- SIDE_ERROR if called with a server.
- 0 if successful.

Example

```
int ret;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_allow_post_handshake_auth(ctx);
if (ret != 0) {
    // failed to allow post handshake authentication
}
```

```
int wolfSSL_allow_post_handshake_auth(
    WOLFSSL * ssl
)
```

This function is called on a TLS v1.3 client wolfSSL to allow a client certificate to be sent post handshake upon request from server. A Post-Handshake Client Authentication extension is sent in the ClientHello. This is useful when connecting to a web server that has some pages that require client authentication and others that don't.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_CTX_allow_post_handshake_auth
- wolfSSL_request_certificate

Return:

- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3.
- SIDE_ERROR if called with a server.
- BAD_STATE_E if called after the handshake has started.
- 0 if successful.

This function must be called before wolfSSL_connect() on the WOLFSSL object.

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_allow_post_handshake_auth(ssl);
if (ret != 0) {
    // failed to allow post handshake authentication
}
```

```
int wolfSSL_CTX_set1_sigalgs_list(
    WOLFSSL_CTX * ctx,
    const char * list
)
```

This function sets the list of signature algorithms to allow on a wolfSSL context in order of preference. The list is a null-terminated text string and a colon-delimited list, where each element is of the form "+" (for example "RSA-PSS+SHA256:ECDSA+SHA384"). The Edwards-curve algorithms "ED25519" and "ED448" are written without a digest suffix, since the hash is implied by the algorithm. The previous list stored in the context is replaced.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().
- **list** a colon-delimited list of "+" elements (or "ED25519" / "ED448" without a digest).

See:

- wolfSSL_set1_sigalgs_list
- wolfSSL_CTX_set1_groups_list

Return:

- WOLFSSL_SUCCESS if successful.

- WOLFSSL_FAILURE if a pointer parameter is NULL, allocation of the suites structure fails, or any token in the list is not recognized or not supported by the current build.

Recognized public-key tokens and digest tokens, together with the build options that must be enabled for each token to be accepted, are listed in the tables below.

Public-key tokens:

Token	Required build option
RSA	!NO_RSA
RSA-PSS / PSS	!NO_RSA and WC_RSA_PSS
ECDSA	HAVE_ECC
ED25519	HAVE_ED25519 (no digest suffix)
ED448	HAVE_ED448 (no digest suffix)
DSA	!NO_DSA
SM2	WOLFSSL_SM2 and WOLFSSL_SM3 (digest is SM3)

Digest tokens:

Token	Required build option
SHA256	!NO_SHA256
SHA384	WOLFSSL_SHA384
SHA512	WOLFSSL_SHA512
SHA224	WOLFSSL_SHA224
SM3	WOLFSSL_SM3
SHA1	!NO_SHA, plus !NO_OLD_TLS or WOLFSSL_ALLOW_TLS_SHA1

Notes for TLS 1.3: per RFC 8446, RSA PKCS#1 v1.5, DSA, SHA-1, and SHA-224 cannot be used as handshake signatures and will be filtered out at negotiation time even if listed. Specifying "RSA-PSS+SHAxxx" causes both the rsa_pss_rsae_shaxxx and rsa_pss_pss_shaxxx schemes to be added. Brainpool ECDSA signature schemes (RFC 8734) cannot be selected through this string interface; they are negotiated automatically when HAVE_ECC_BRAINPOOL is enabled.

Example

```
int ret;
WOLFSSL_CTX* ctx;
const char* list = "RSA-PSS+SHA256:ECDSA+SHA384:ED25519";
...
ret = wolfSSL_CTX_set1_sigalgs_list(ctx, list);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set signature algorithm list
}
```

```
int wolfSSL_set1_sigalgs_list(
    WOLFSSL * ssl,
    const char * list
)
```

This function sets the list of signature algorithms to allow on a wolfSSL session in order of preference. The list format and the set of recognized public-key and digest tokens are identical to those documented for wolfSSL_CTX_set1_sigalgs_list(); refer to that function for the full token tables and TLS 1.3 caveats. The previous list stored in the session is replaced.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **list** a colon-delimited list of "+" elements (or "ED25519" / "ED448" without a digest).

See:

- wolfSSL_CTX_set1_sigalgs_list
- wolfSSL_set1_groups_list

Return:

- WOLFSSL_SUCCESS if successful.
- WOLFSSL_FAILURE if a pointer parameter is NULL, allocation of the suites structure fails, or any token in the list is not recognized or not supported by the current build.

Example

```
int ret;
WOLFSSL* ssl;
const char* list = "RSA-PSS+SHA256:ECDSA+SHA384:ED25519";
...
ret = wolfSSL_set1_sigalgs_list(ssl, list);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set signature algorithm list
}
```

```
int wolfSSL_CTX_set1_groups_list(
    WOLFSSL_CTX * ctx,
    const char * list
)
```

This function sets the list of key-exchange groups (named elliptic curves and KEMs) to allow on a wolfSSL context in order of preference. The list is a null-terminated, colon-delimited text string of group names, for example "P-384:P-256:X25519". Call this function to set the key-exchange parameters used with TLS v1.3 connections (the function is compiled in only when HAVE_ECC, WOLFSSL_TLS13, and HAVE_SUPPORTED_CURVES are defined).

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().
- **list** a string that is a colon-delimited list of key-exchange group names.

See:

- wolfSSL_set1_groups_list
- wolfSSL_CTX_set_groups
- wolfSSL_set_groups
- wolfSSL_UseKeyShare
- wolfSSL_preferred_group

Return:

- WOLFSSL_FAILURE if pointer parameters are NULL, there are more than WOLFSSL_MAX_GROUP_COUNT groups, a group name is not recognized or not using TLS v1.3.
- WOLFSSL_SUCCESS if successful.

Recognized group names and the build options each one requires are listed below. Names are matched case-sensitively against the table.

NIST / SEC curves (require HAVE_ECC):

Name	Curve / Group
P-160	secp160r1
P-160-2	secp160r2
P-192	secp192r1 (prime192v1)
P-224	secp224r1
P-256	secp256r1 (prime256v1) — also accepted as “prime256v1” / “secp256r1”
P-384	secp384r1 — also accepted as “secp384r1”
P-521	secp521r1 — also accepted as “secp521r1”
K-160	secp160k1
K-192	secp192k1
K-224	secp224k1
K-256	secp256k1

Brainpool curves (require HAVE_ECC plus WOLFSSL_CUSTOM_CURVES and HAVE_ECC_BRAINPOOL — typically enabled by –enable-ecccustcurves=all):

Name	Curve
B-256	brainpoolP256r1
B-384	brainpoolP384r1
B-512	brainpoolP512r1

Edwards / Montgomery curves:

Name	Required build option
X25519	HAVE_CURVE25519
X448	HAVE_CURVE448

SM2 (requires WOLFSSL_SM2):

Name	Group
SM2	sm2p256v1
sm2p256v1	sm2p256v1 (alias)

ML-KEM (post-quantum) groups (require WOLFSSL_HAVE_MLKEM and !WOLFSSL_NO_ML_KEM):

Name
ML_KEM_512
ML_KEM_768
ML_KEM_1024

ML-KEM hybrid groups additionally require HAVE_ECC together with either WOLFSSL_WC_MLKEM, and WOLFSSL_PQC_HYBRIDS (or WOLFSSL_EXTRA_PQC_HYBRIDS for the “extra” set):

Name	Hybrid flag set
SecP256r1MLKEM768	WOLFSSL_PQC_HYBRIDS
SecP384r1MLKEM1024	WOLFSSL_PQC_HYBRIDS
X25519MLKEM768	WOLFSSL_PQC_HYBRIDS
SecP256r1MLKEM512	WOLFSSL_EXTRA_PQC_HYBRIDS
SecP384r1MLKEM768	WOLFSSL_EXTRA_PQC_HYBRIDS
SecP521r1MLKEM1024	WOLFSSL_EXTRA_PQC_HYBRIDS
X25519MLKEM512	WOLFSSL_EXTRA_PQC_HYBRIDS
X448MLKEM768	WOLFSSL_EXTRA_PQC_HYBRIDS

Legacy Kyber groups (require WOLFSSL_MLKEM_KYBER; hybrids additionally require HAVE_ECC together with WOLFSSL_WC_MLKEM):

Name
KYBER_LEVEL1
KYBER_LEVEL3
KYBER_LEVEL5
P256_KYBER_LEVEL1
P256_KYBER_LEVEL3
P384_KYBER_LEVEL3
P521_KYBER_LEVEL5
X25519_KYBER_LEVEL1
X25519_KYBER_LEVEL3
X448_KYBER_LEVEL3

In addition to the names above, when HAVE_FIPS and HAVE_SELFTEST are not defined, any curve name registered with wolfCrypt (looked up via wc_ecc_get_curve_idx_from_name(), e.g. “brainpoolP256r1”) is also accepted.

The order of the names in the list is preserved and used as the local preference order for KeyShare selection in TLS 1.3.

Example

```
int ret;
WOLFSSL_CTX* ctx;
const char* list = "P-384:P-256";
...
ret = wolfSSL_CTX_set1_groups_list(ctx, list);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set group list
}
```

```
int wolfSSL_set1_groups_list(
    WOLFSSL * ssl,
    const char * list
)
```

This function sets the list of key-exchange groups (named elliptic curves and KEMs) to allow on a wolfSSL session in order of preference. The list format and the set of recognized group names are identical to those documented for wolfSSL_CTX_set1_groups_list(); refer to that function for the full token tables and required build options.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **list** a string that is a colon separated list of key exchange groups.

See:

- wolfSSL_CTX_set1_groups_list
- wolfSSL_CTX_set_groups
- wolfSSL_set_groups

- wolfSSL_UseKeyShare
- wolfSSL_preferred_group

Return:

- WOLFSSL_FAILURE if pointer parameters are NULL, there are more than WOLFSSL_MAX_GROUP_COUNT groups, a group name is not recognized or not using TLS v1.3.
- WOLFSSL_SUCCESS if successful.

Example

```
int ret;
WOLFSSL* ssl;
const char* list = "P-384:P-256";
...
ret = wolfSSL_CTX_set1_groups_list(ssl, list);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set group list
}
```

```
int wolfSSL_CTX_set_groups(
    WOLFSSL_CTX * ctx,
    int * groups,
    int count
)
```

This function sets the list of key-exchange groups (named elliptic curves and KEMs) to allow on a wolfSSL context in order of preference. The list is an array of named-group identifiers (see the table below) and `count` is the number of identifiers in the array. Use this function to set the key-exchange parameters used by TLS v1.3 connections; the order of the array becomes the local KeyShare preference order.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().
- **groups** a list of key-exchange groups by identifier.
- **count** the number of identifiers in `groups` (must not exceed WOLFSSL_MAX_GROUP_COUNT).

See:

- wolfSSL_set_groups
- wolfSSL_UseKeyShare
- wolfSSL_CTX_set1_groups_list
- wolfSSL_set1_groups_list
- wolfSSL_preferred_group

Return:

- BAD_FUNC_ARG if a pointer parameter is NULL, `count` exceeds WOLFSSL_MAX_GROUP_COUNT, or the underlying method is not a TLS method.

- WOLFSSL_SUCCESS if successful.

Recognized identifiers and the build options each one requires are listed below. The identifiers are defined in the anonymous enum in <wolfssl/ssl.h>.

NIST / SEC curves (require HAVE_ECC):

Identifier	Curve / Group
WOLFSSL_ECC_SECP160K1	secp160k1
WOLFSSL_ECC_SECP160R1	secp160r1
WOLFSSL_ECC_SECP160R2	secp160r2
WOLFSSL_ECC_SECP192K1	secp192k1
WOLFSSL_ECC_SECP192R1	secp192r1 (prime192v1)
WOLFSSL_ECC_SECP224K1	secp224k1
WOLFSSL_ECC_SECP224R1	secp224r1
WOLFSSL_ECC_SECP256K1	secp256k1
WOLFSSL_ECC_SECP256R1	secp256r1 (prime256v1)
WOLFSSL_ECC_SECP384R1	secp384r1
WOLFSSL_ECC_SECP521R1	secp521r1

Brainpool curves (require HAVE_ECC plus WOLFSSL_CUSTOM_CURVES and HAVE_ECC_BRAINPOOL — typically enabled by –enable-ecccustcurves=all):

Identifier	Curve	Notes
WOLFSSL_ECC_BRAINPOOLP256R1	brainpoolP256r1	TLS 1.2 group ID 26
WOLFSSL_ECC_BRAINPOOLP384R1	brainpoolP384r1	TLS 1.2 group ID 27
WOLFSSL_ECC_BRAINPOOLP512R1	brainpoolP512r1	TLS 1.2 group ID 28
WOLFSSL_ECC_BRAINPOOLP256R1TLS13	brainpoolP256r1	RFC 8734 TLS 1.3 ID
WOLFSSL_ECC_BRAINPOOLP384R1TLS13	brainpoolP384r1	RFC 8734 TLS 1.3 ID
WOLFSSL_ECC_BRAINPOOLP512R1TLS13	brainpoolP512r1	RFC 8734 TLS 1.3 ID

Edwards / Montgomery curves:

Identifier	Required build option
WOLFSSL_ECC_X25519	HAVE_CURVE25519
WOLFSSL_ECC_X448	HAVE_CURVE448

SM2 (requires WOLFSSL_SM2):

Identifier	Group
WOLFSSL_ECC_SM2P256V1	sm2p256v1

Finite-field DH (RFC 7919) groups (require HAVE_FFDHE and the matching HAVE_FFDHE_NNNN macro for each size):

Identifier	Group
WOLFSSL_FFDHE_2048	ffdhe2048
WOLFSSL_FFDHE_3072	ffdhe3072
WOLFSSL_FFDHE_4096	ffdhe4096
WOLFSSL_FFDHE_6144	ffdhe6144
WOLFSSL_FFDHE_8192	ffdhe8192

ML-KEM (post-quantum) groups (require HAVE_PQC, WOLFSSL_HAVE_MLKEM and !WOLFSSL_NO_ML_KEM):

Identifier
WOLFSSL_ML_KEM_512
WOLFSSL_ML_KEM_768
WOLFSSL_ML_KEM_1024

ML-KEM hybrid groups additionally require HAVE_ECC together with either WOLFSSL_WC_MLKEM, and WOLFSSL_PQC_HYBRIDS (or WOLFSSL_EXTRA_PQC_HYBRIDS for the “extra” set):

Identifier	Hybrid flag set
WOLFSSL_SECP256R1MLKEM768	WOLFSSL_PQC_HYBRIDS
WOLFSSL_X25519MLKEM768	WOLFSSL_PQC_HYBRIDS
WOLFSSL_SECP384R1MLKEM1024	WOLFSSL_PQC_HYBRIDS
WOLFSSL_SECP256R1MLKEM512	WOLFSSL_EXTRA_PQC_HYBRIDS
WOLFSSL_SECP384R1MLKEM768	WOLFSSL_EXTRA_PQC_HYBRIDS
WOLFSSL_SECP521R1MLKEM1024	WOLFSSL_EXTRA_PQC_HYBRIDS
WOLFSSL_X25519MLKEM512	WOLFSSL_EXTRA_PQC_HYBRIDS
WOLFSSL_X448MLKEM768	WOLFSSL_EXTRA_PQC_HYBRIDS

Legacy Kyber groups (require HAVE_PQC and WOLFSSL_MLKEM_KYBER; hybrids additionally require HAVE_ECC together with WOLFSSL_WC_MLKEM):

Identifier
WOLFSSL_KYBER_LEVEL1
WOLFSSL_KYBER_LEVEL3
WOLFSSL_KYBER_LEVEL5
WOLFSSL_P256_KYBER_LEVEL1
WOLFSSL_P256_KYBER_LEVEL3
WOLFSSL_P384_KYBER_LEVEL3
WOLFSSL_P521_KYBER_LEVEL5
WOLFSSL_X25519_KYBER_LEVEL1
WOLFSSL_X25519_KYBER_LEVEL3
WOLFSSL_X448_KYBER_LEVEL3

Example

```
int ret;
WOLFSSL_CTX* ctx;
int groups[] = { WOLFSSL_ECC_X25519, WOLFSSL_ECC_SECP256R1 };
int count = sizeof(groups) / sizeof(groups[0]);
...
ret = wolfSSL_CTX_set_groups(ctx, groups, count);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set group list
}
```

```
int wolfSSL_set_groups(
    WOLFSSL * ssl,
    int * groups,
    int count
)
```

This function sets the list of key-exchange groups (named elliptic curves and KEMs) to allow on a wolfSSL session in order of preference. The array format and the set of recognized identifiers are identical to those documented for wolfSSL_CTX_set_groups(); refer to that function for the full identifier table and required build options.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **groups** a list of key-exchange groups by identifier.
- **count** the number of identifiers in `groups` (must not exceed WOLFSSL_MAX_GROUP_COUNT).

See:

- wolfSSL_CTX_set_groups
- wolfSSL_UseKeyShare
- wolfSSL_CTX_set1_groups_list
- wolfSSL_set1_groups_list
- wolfSSL_preferred_group

Return:

- BAD_FUNC_ARG if a pointer parameter is NULL, count exceeds WOLFSSL_MAX_GROUP_COUNT, any of the identifiers are unrecognized, or the underlying method is not a TLS method.
- WOLFSSL_SUCCESS if successful.

Example

```
int ret;
WOLFSSL* ssl;
int groups[] = { WOLFSSL_ECC_X25519, WOLFSSL_ECC_SECP256R1 };
int count = sizeof(groups) / sizeof(groups[0]);
...
ret = wolfSSL_set_groups(ssl, groups, count);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set group list
}
```

```
int wolfSSL_CTX_set_max_early_data(
    WOLFSSL_CTX * ctx,
    unsigned int sz
)
```

This function sets the maximum amount of early data that a TLS v1.3 client or server is willing to exchange using the wolfSSL context. Call this function to limit the amount of early data to process to mitigate replay attacks. Early data is protected by keys derived from those of the connection that the session ticket was sent and therefore will be the same every time a session ticket is used in resumption. The value is included in the session ticket for resumption. A server value of zero indicates no early data is to be sent by client using session tickets. A client value of zero indicates that the client will not send any early data. The default value is zero: per RFC 8446 Appendix E.5, TLS implementations "MUST NOT enable 0-RTT (either sending or accepting) unless specifically requested by the application." Servers must call this function (or the per-SSL equivalent) with a non-zero value to opt in. It is recommended that the number of early data bytes be kept as low as practically possible in the application.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().
- **sz** the amount of early data to accept in bytes.

See:

- wolfSSL_set_max_early_data
- wolfSSL_write_early_data
- wolfSSL_read_early_data

Return:

- BAD_FUNC_ARG if ctx is NULL or not using TLS v1.3.
- 0 if successful.

Example

```
int ret;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_set_max_early_data(ctx, 128);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set group list
}
```

```
int wolfSSL_set_max_early_data(
    WOLFSSL * ssl,
    unsigned int sz
)
```

This function sets the maximum amount of early data that a TLS v1.3 client or server is willing to exchange. Call this function to limit the amount of early data to process to mitigate replay attacks. Early data is protected by keys derived from those of the connection that the session ticket was sent and therefore will be the same every time a session ticket is used in resumption. The value is included in the session ticket for resumption. A server value of zero indicates no early data is to be sent by client using session tickets. A client value of zero indicates that the client will not send any early data. It is recommended that the number of early data bytes be kept as low as practically possible in the application.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **sz** the amount of early data to accept from client in bytes.

See:

- wolfSSL_CTX_set_max_early_data
- wolfSSL_write_early_data
- wolfSSL_read_early_data

Return:

- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3.
- 0 if successful.

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_set_max_early_data(ssl, 128);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set group list
}
```

```
int wolfSSL_CTX_no_early_data_fresh_start_check(
    WOLFSSL_CTX * ctx
)
```

This function is called on the server to disable the RFC 8446 Section 8.2 fresh start protection. By default a freshly created context rejects early data, but not resumption, for session tickets minted before the context was created, since the anti-replay state for those tickets may not have survived a server restart. Only call this function when the anti-replay state (session cache or external cache) reliably survives server restarts.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().

See:

- wolfSSL_CTX_set_max_early_data
- wolfSSL_read_early_data

Return:

- BAD_FUNC_ARG if ctx is NULL or not using TLS v1.3.
- SIDE_ERROR if called with a client.
- 0 if successful.

The check compares the ticket timestamp against the context creation time, both taken from TimeNowInMilliseconds(). That clock is only required to be millisecond accurate, not correlated to the epoch, so on ports where it counts from boot (Windows QPC, Zephyr, FreeRTOS, Micrium, Microchip) it restarts near zero and the check does not fire for tickets minted before a reboot. Such deployments must rely on ticket key rotation instead.

Example

```
int ret;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_no_early_data_fresh_start_check(ctx);
if (ret != 0) {
    // failed to disable the fresh start check
}
```

```
void wolfSSL_CTX_set_psk_client_tls13_callback(
    WOLFSSL_CTX * ctx,
    wc_psk_client_tls13_callback cb
)
```

This function sets the Pre-Shared Key (PSK) client side callback for TLS v1.3 connections. The callback is used to find a PSK identity and return its key and the name of the cipher to use for the handshake. The function sets the client_psk_tls13_cb member of the WOLFSSL_CTX structure.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().
- **cb** a Pre-Shared Key (PSK) callback for a TLS 1.3 client.

See:

- wolfSSL_set_psk_client_tls13_callback
- wolfSSL_CTX_set_psk_server_tls13_callback
- wolfSSL_set_psk_server_tls13_callback

Example

```
WOLFSSL_CTX* ctx;
...
wolfSSL_CTX_set_psk_client_tls13_callback(ctx, my_psk_client_tls13_cb);
```

```
void wolfSSL_set_psk_client_tls13_callback(
    WOLFSSL * ssl,
    wc_psk_client_tls13_callback cb
)
```

This function sets the Pre-Shared Key (PSK) client side callback for TLS v1.3 connections. The callback is used to find a PSK identity and return its key and the name of the cipher to use for the handshake. The function sets the client_psk_tls13_cb member of the options field in WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **cb** a Pre-Shared Key (PSK) callback for a TLS 1.3 client.

See:

- wolfSSL_CTX_set_psk_client_tls13_callback
- wolfSSL_CTX_set_psk_server_tls13_callback
- wolfSSL_set_psk_server_tls13_callback

Example

```
WOLFSSL* ssl;
...
wolfSSL_set_psk_client_tls13_callback(ssl, my_psk_client_tls13_cb);
```

```
void wolfSSL_CTX_set_psk_server_tls13_callback(
    WOLFSSL_CTX * ctx,
    wc_psk_server_tls13_callback cb
)
```

This function sets the Pre-Shared Key (PSK) server side callback for TLS v1.3 connections. The callback is used to find a PSK identity and return its key and the name of the cipher to use for the handshake. The function sets the server_psk_tls13_cb member of the WOLFSSL_CTX structure.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().
- **cb** a Pre-Shared Key (PSK) callback for a TLS 1.3 server.

See:

- wolfSSL_CTX_set_psk_client_tls13_callback
- wolfSSL_set_psk_client_tls13_callback
- wolfSSL_set_psk_server_tls13_callback

Example

```
WOLFSSL_CTX* ctx;
...
wolfSSL_CTX_set_psk_server_tls13_callback(ctx, my_psk_client_tls13_cb);
```

```
void wolfSSL_set_psk_server_tls13_callback(
    WOLFSSL * ssl,
    wc_psk_server_tls13_callback cb
)
```

This function sets the Pre-Shared Key (PSK) server side callback for TLS v1.3 connections. The callback is used to find a PSK identity and return its key and the name of the cipher to use for the handshake. The function sets the server_psk_tls13_cb member of the options field in WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **cb** a Pre-Shared Key (PSK) callback for a TLS 1.3 server.

See:

- wolfSSL_CTX_set_psk_client_tls13_callback
- wolfSSL_set_psk_client_tls13_callback
- wolfSSL_CTX_set_psk_server_tls13_callback

Example

```
WOLFSSL* ssl;
...
wolfSSL_set_psk_server_tls13_callback(ssl, my_psk_server_tls13_cb);
```

```
int wolfSSL_CTX_set_cert_with_extern_psk(
    WOLFSSL_CTX * ctx,
    int state
)
```

Enable or disable TLS 1.3 certificate authentication with external PSK (RFC 9973) on a context.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().
- **state** 0 to disable, non-zero to enable.

See:

- wolfSSL_set_cert_with_extern_psk
- wolfSSL_CTX_set_psk_client_tls13_callback
- wolfSSL_CTX_set_psk_server_tls13_callback

Return:

- WOLFSSL_SUCCESS on success.
- WOLFSSL_FAILURE when `ctx` is NULL.

When enabled, wolfSSL advertises and accepts the `tls_cert_with_extern_psk` extension for TLS 1.3 handshakes using external PSKs. Any non-zero `state` value enables the feature and zero disables it.

Availability:

- Built with `--enable-tls13 --enable-psk --enable-cert-with-extern-psk`
- Or with `WOLFSSL_TLS13` and `WOLFSSL_CERT_WITH_EXTERN_PSK` defined

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(wolfTLSv1_3_client_method());
if (wolfSSL_CTX_set_cert_with_extern_psk(ctx, 1) != WOLFSSL_SUCCESS) {
    /* handle error */
}
```

```
int wolfSSL_set_cert_with_extern_psk(
    WOLFSSL * ssl,
    int state
)
```

Enable or disable TLS 1.3 certificate authentication with external PSK (RFC 9973) on a connection.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **state** 0 to disable, non-zero to enable.

See:

- wolfSSL_CTX_set_cert_with_extern_psk
- wolfSSL_set_psk_client_tls13_callback
- wolfSSL_set_psk_server_tls13_callback

Return:

- WOLFSSL_SUCCESS on success.
- WOLFSSL_FAILURE when `ssl` is NULL.

This call applies to a single WOLFSSL object. Any non-zero `state` value enables the feature and zero disables it.

Availability:

- Built with `--enable-tls13 --enable-psk --enable-cert-with-extern-psk`
- Or with `WOLFSSL_TLS13` and `WOLFSSL_CERT_WITH_EXTERN_PSK` defined

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
if (wolfSSL_set_cert_with_extern_psk(ssl, 1) != WOLFSSL_SUCCESS) {
    /* handle error */
}
```

```
int wolfSSL_UseKeyShare(
    WOLFSSL * ssl,
    word16 group
)
```

This function creates a key share entry from the group including generating a key pair. The KeyShare extension contains all the generated public keys for key exchange. If this function is called, then only the groups specified will be included. Call this function when a preferred group has been previously established for the server.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **group** a key exchange group identifier.

See:

- wolfSSL_preferred_group
- wolfSSL_CTX_set1_groups_list
- wolfSSL_set1_groups_list
- wolfSSL_CTX_set_groups
- wolfSSL_set_groups
- wolfSSL_NoKeyShares

Return:

- BAD_FUNC_ARG if ssl is NULL.
- MEMORY_E when dynamic memory allocation fails.
- WOLFSSL_SUCCESS if successful.

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_UseKeyShare(ssl, WOLFSSL_ECC_X25519);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set key share
}
```

```
int wolfSSL_NoKeyShares(
    WOLFSSL * ssl
)
```

This function is called to ensure no key shares are sent in the ClientHello. This will force the server to respond with a HelloRetryRequest if a key exchange is required in the handshake. Call this function when the expected key exchange group is not known and to avoid the generation of keys unnecessarily. Note that an extra round-trip will be required to complete the handshake when a key exchange is required.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_UseKeyShare

Return:

- BAD_FUNC_ARG if ssl is NULL.
- SIDE_ERROR if called with a server.
- WOLFSSL_SUCCESS if successful.

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_NoKeyShares(ssl);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set no key shares
}
```

```
WOLFSSL_METHOD * wolfTLSv1_3_server_method_ex(
    void * heap
)
```

This function is used to indicate that the application is a server and will only support the TLS 1.3 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Parameters:

- **heap** a pointer to a buffer that the static memory allocator will use during dynamic memory allocation.

See:

- wolfSSLv3_server_method
- wolfTLSv1_server_method
- wolfTLSv1_1_server_method
- wolfTLSv1_2_server_method
- wolfTLSv1_3_server_method
- wolfDTLSv1_server_method
- wolfSSLv23_server_method
- wolfSSL_CTX_new

Return:

- If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_3_server_method_ex(NULL);
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

```
WOLFSSL_METHOD * wolfTLSv1_3_client_method_ex(
    void * heap
)
```

This function is used to indicate that the application is a client and will only support the TLS 1.3 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Parameters:

- **heap** a pointer to a buffer that the static memory allocator will use during dynamic memory allocation.

See:

- wolfSSLv3_client_method
- wolfTLSv1_client_method
- wolfTLSv1_1_client_method
- wolfTLSv1_2_client_method
- wolfTLSv1_3_client_method
- wolfDTLSv1_client_method
- wolfSSLv23_client_method
- wolfSSL_CTX_new

Return:

- If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_3_client_method_ex(NULL);
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

```
WOLFSSL_METHOD * wolfTLSv1_3_server_method(
    void
)
```

This function is used to indicate that the application is a server and will only support the TLS 1.3 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

See:

- wolfSSLv3_server_method
- wolfTLSv1_server_method
- wolfTLSv1_1_server_method
- wolfTLSv1_2_server_method
- wolfTLSv1_3_server_method_ex
- wolfDTLSv1_server_method
- wolfSSLv23_server_method
- wolfSSL_CTX_new

Return:

- If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_3_server_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

```
WOLFSSL_METHOD * wolfTLSv1_3_client_method(
    void
)
```

This function is used to indicate that the application is a client and will only support the TLS 1.3 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

See:

- wolfSSLv3_client_method
- wolfTLSv1_client_method
- wolfTLSv1_1_client_method
- wolfTLSv1_2_client_method
- wolfTLSv1_3_client_method_ex
- wolfDTLSv1_client_method
- wolfSSLv23_client_method
- wolfSSL_CTX_new

Return:

- If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_3_client_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

```
WOLFSSL_METHOD * wolfTLSv1_3_method_ex(
    void * heap
)
```

This function returns a WOLFSSL_METHOD similar to wolfTLSv1_3_client_method except that it is not determined which side yet (server/client).

Parameters:

- **heap** a pointer to a buffer that the static memory allocator will use during dynamic memory allocation.

See:

- wolfSSL_new
- wolfSSL_free

Return:

- WOLFSSL_METHOD On successful creations returns a WOLFSSL_METHOD pointer
- NULL Null if memory allocation error or failure to create method

Example

```
WOLFSSL* ctx;
ctx  = wolfSSL_CTX_new(wolfTLSv1_3_method_ex(NULL));
// check ret value
```

```
WOLFSSL_METHOD * wolfTLSv1_3_method(
    void
)
```

This function returns a WOLFSSL_METHOD similar to wolfTLSv1_3_client_method except that it is not determined which side yet (server/client).

See:

- wolfSSL_new
- wolfSSL_free

Return:

- WOLFSSL_METHOD On successful creations returns a WOLFSSL_METHOD pointer
- NULL Null if memory allocation error or failure to create method

Example

```
WOLFSSL* ctx;
ctx  = wolfSSL_CTX_new(wolfTLSv1_3_method());
// check ret value
```

```
int wolfSSL_CTX_set_client_cert_type(
    WOLFSSL_CTX * ctx,
    const char * buf,
    int len
)
```

In case this function is called in a client side, set certificate types that can be sent to its peer. In case called in a server side, set certificate types that can be acceptable from its peer. Put cert types in the buffer with prioritised order. To reset the settings to default, pass NULL for the buffer or pass zero for len. By default, certificate type is only X509. In case both side intend to send or accept “Raw public key” cert, WOLFSSL_CERT_TYPE_RPK should be included in the buffer to set.

Parameters:

- **ctx** WOLFSSL_CTX object pointer
- **buf** A buffer where certificate types are stored
- **len** buf size in bytes (same as number of certificate types included) *Example*

```
int ret;
WOLFSSL_CTX* ctx;
char buf[] = {WOLFSSL_CERT_TYPE_RPK, WOLFSSL_CERT_TYPE_X509};
int len = sizeof(buf)/sizeof(char);
...

ret = wolfSSL_CTX_set_client_cert_type(ctx, buf, len);
```

See:

- wolfSSL_set_client_cert_type
- wolfSSL_CTX_set_server_cert_type
- wolfSSL_set_server_cert_type
- wolfSSL_get_negotiated_client_cert_type
- wolfSSL_get_negotiated_server_cert_type

Return:

- WOLFSSL_SUCCESS if cert types set successfully
- BAD_FUNC_ARG if NULL was passed for ctx, illegal value was specified as cert type, buf size exceed MAX_CLIENT_CERT_TYPE_CNT was specified or a duplicate value is found in buf.

```
int wolfSSL_CTX_set_server_cert_type(
    WOLFSSL_CTX * ctx,
    const char * buf,
    int len
)
```

In case this function is called in a server side, set certificate types that can be sent to its peer. In case called in a client side, set certificate types that can be acceptable from its peer. Put cert types in the buffer with prioritised order. To reset the settings to default, pass NULL for the buffer or pass zero for len. By default, certificate type is only X509. In case both side intend to send or accept “Raw public key” cert, WOLFSSL_CERT_TYPE_RPK should be included in the buffer to set.

Parameters:

- **ctx** WOLFSSL_CTX object pointer
- **buf** A buffer where certificate types are stored
- **len** buf size in bytes (same as number of certificate types included) *Example*

```
int ret;
WOLFSSL_CTX* ctx;
char buf[] = {WOLFSSL_CERT_TYPE_RPK, WOLFSSL_CERT_TYPE_X509};
int len = sizeof(buf)/sizeof(char);
...

ret = wolfSSL_CTX_set_server_cert_type(ctx, buf, len);
```

See:

- wolfSSL_set_client_cert_type
- wolfSSL_CTX_set_client_cert_type
- wolfSSL_set_server_cert_type
- wolfSSL_get_negotiated_client_cert_type
- wolfSSL_get_negotiated_server_cert_type

Return:

- WOLFSSL_SUCCESS if cert types set successfully
- BAD_FUNC_ARG if NULL was passed for ctx, illegal value was specified as cert type, buf size exceed MAX_SERVER_CERT_TYPE_CNT was specified or a duplicate value is found in buf.

```
int wolfSSL_set_client_cert_type(
    WOLFSSL * ssl,
    const char * buf,
    int len
)
```

In case this function is called in a client side, set certificate types that can be sent to its peer. In case called in a server side, set certificate types that can be acceptable from its peer. Put cert types in the buffer with prioritised order. To reset the settings to default, pass NULL for the buffer or pass zero for len. By default, certificate type is only X509. In case both side intend to send or accept "Raw public key" cert, WOLFSSL_CERT_TYPE_RPK should be included in the buffer to set.

Parameters:

- **ssl** WOLFSSL object pointer
- **buf** A buffer where certificate types are stored
- **len** buf size in bytes (same as number of certificate types included) *Example*

```
int ret;
WOLFSSL* ssl;
char buf[] = {WOLFSSL_CERT_TYPE_RPK, WOLFSSL_CERT_TYPE_X509};
int len = sizeof(buf)/sizeof(char);
...

ret = wolfSSL_set_client_cert_type(ssl, buf, len);
```

See:

- wolfSSL_CTX_set_client_cert_type
- wolfSSL_CTX_set_server_cert_type
- wolfSSL_set_server_cert_type
- wolfSSL_get_negotiated_client_cert_type
- wolfSSL_get_negotiated_server_cert_type

Return:

- WOLFSSL_SUCCESS if cert types set successfully
- BAD_FUNC_ARG if NULL was passed for ctx, illegal value was specified as cert type, buf size exceed MAX_CLIENT_CERT_TYPE_CNT was specified or a duplicate value is found in buf.

```
int wolfSSL_set_server_cert_type(
    WOLFSSL * ssl,
    const char * buf,
    int len
)
```

In case this function is called in a server side, set certificate types that can be sent to its peer. In case called in a client side, set certificate types that can be acceptable from its peer. Put cert types in the buffer with prioritised order. To reset the settings to default, pass NULL for the buffer or pass zero for len. By default, certificate type is only X509. In case both side intend to send or accept "Raw public key" cert, WOLFSSL_CERT_TYPE_RPK should be included in the buffer to set.

Parameters:

- **ctx** WOLFSSL_CTX object pointer
- **buf** A buffer where certificate types are stored
- **len** buf size in bytes (same as number of certificate types included) *Example*

```
int ret;
WOLFSSL* ssl;
char buf[] = {WOLFSSL_CERT_TYPE_RPK, WOLFSSL_CERT_TYPE_X509};
int len = sizeof(buf)/sizeof(char);
...

ret = wolfSSL_set_server_cert_type(ssl, buf, len);
```

See:

- wolfSSL_set_client_cert_type
- wolfSSL_CTX_set_server_cert_type
- wolfSSL_set_server_cert_type
- wolfSSL_get_negotiated_client_cert_type
- wolfSSL_get_negotiated_server_cert_type

Return:

- WOLFSSL_SUCCESS if cert types set successfully
- BAD_FUNC_ARG if NULL was passed for ctx, illegal value was specified as cert type, buf size exceed MAX_SERVER_CERT_TYPE_CNT was specified or a duplicate value is found in buf.

```
int wolfSSL_CTX_clear_group_messages(
    WOLFSSL_CTX * ctx
)
```

Disables handshake message grouping for the given WOLFSSL_CTX context.

Parameters:

- **ctx** Pointer to the WOLFSSL_CTX structure.

See:

- wolfSSL_CTX_set_group_messages
- wolfSSL_set_group_messages
- wolfSSL_clear_group_messages

Return:

- WOLFSSL_SUCCESS on success.
- BAD_FUNC_ARG if ctx is NULL.

This function turns off handshake message grouping for all SSL objects created from the specified context.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(wolfTLSv1_2_client_method());
wolfSSL_CTX_clear_group_messages(ctx);
```

```
int wolfSSL_clear_group_messages(
    WOLFSSL * ssl
)
```

Disables handshake message grouping for the given WOLFSSL object.

Parameters:

- **ssl** Pointer to the WOLFSSL structure.

See:

- wolfSSL_set_group_messages
- wolfSSL_CTX_set_group_messages
- wolfSSL_CTX_clear_group_messages

Return:

- WOLFSSL_SUCCESS on success.
- BAD_FUNC_ARG if ssl is NULL.

This function turns off handshake message grouping for the specified SSL object.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
wolfSSL_clear_group_messages(ssl);
```

```
int wolfSSL_CTX_set_expected_rpk(
    WOLFSSL_CTX * ctx,
    const unsigned char * spki,
    unsigned int spkiSz
)
```

Pin a DER-encoded SubjectPublicKeyInfo that the peer is expected to present as a Raw Public Key (RFC 7250), establishing out-of-band trust on the WOLFSSL_CTX object. An unauthenticated RPK peer is otherwise rejected: when the peer is being authenticated (any verify mode other than WOLFSSL_VERIFY_NONE) the handshake fails closed unless the presented key matches a pin (or a verify callback accepts it). May be called more than once to pin several keys, up to WOLFSSL_MAX_RPK_PINS. The key is stored as its SHA-256 digest, so this API requires SHA-256. Pins are append-only - there is no per-entry remove, but wolfSSL_CTX_clear_expected_rpk() empties the table so it can be repopulated.

Parameters:

- **ctx** WOLFSSL_CTX object pointer
- **spki** DER-encoded SubjectPublicKeyInfo the peer is expected to present
- **spkiSz** length of spki in bytes *Example*

```
int ret;
WOLFSSL_CTX* ctx;
const unsigned char* spki;
unsigned int spkiSz;
...

ret = wolfSSL_CTX_set_expected_rpk(ctx, spki, spkiSz);
```

See:

- wolfSSL_set_expected_rpk
- wolfSSL_CTX_clear_expected_rpk
- wolfSSL_set_client_cert_type
- wolfSSL_set_server_cert_type

Return:

- WOLFSSL_SUCCESS on success
- BAD_FUNC_ARG if ctx or spki is NULL, or spkiSz is 0
- BUFFER_E if the pin table is already full (WOLFSSL_MAX_RPK_PINS)
- other negative error code on a hashing failure

```
int wolfSSL_set_expected_rpk(
    WOLFSSL * ssl,
    const unsigned char * spki,
    unsigned int spkiSz
)
```

Pin a DER-encoded SubjectPublicKeyInfo that the peer is expected to present as a Raw Public Key (RFC 7250), establishing out-of-band trust on the WOLFSSL object. An unauthenticated RPK peer

is otherwise rejected: when the peer is being authenticated (any verify mode other than WOLFSSL_VERIFY_NONE) the handshake fails closed unless the presented key matches a pin (or a verify callback accepts it). May be called more than once to pin several keys, up to WOLFSSL_MAX_RPK_PINS. The key is stored as its SHA-256 digest, so this API requires SHA-256. Pins are append-only - there is no per-entry remove, but wolfSSL_clear_expected_rpk() empties the table so it can be repopulated.

Parameters:

- **ssl** WOLFSSL object pointer
- **spki** DER-encoded SubjectPublicKeyInfo the peer is expected to present
- **spkiSz** length of spki in bytes *Example*

```
int ret;
WOLFSSL* ssl;
const unsigned char* spki;
unsigned int spkiSz;
...

ret = wolfSSL_set_expected_rpk(ssl, spki, spkiSz);
```

See:

- wolfSSL_CTX_set_expected_rpk
- wolfSSL_clear_expected_rpk
- wolfSSL_set_client_cert_type
- wolfSSL_set_server_cert_type

Return:

- WOLFSSL_SUCCESS on success
- BAD_FUNC_ARG if ssl or spki is NULL, or spkiSz is 0
- BUFFER_E if the pin table is already full (WOLFSSL_MAX_RPK_PINS)
- other negative error code on a hashing failure

```
int wolfSSL_CTX_clear_expected_rpk(
    WOLFSSL_CTX * ctx
)
```

Remove all pinned expected peer Raw Public Keys (RFC 7250) from the WOLFSSL_CTX object, emptying the table so it can be repopulated - for example across a peer key rotation, since the pinning APIs are otherwise append-only. Like other WOLFSSL_CTX setters this is not locked, so reconfigure the pins on a shared CTX while no handshakes are in flight (a WOLFSSL created with wolfSSL_new() copies the pin table by value at creation time).

Parameters:

- **ctx** WOLFSSL_CTX object pointer *Example*

```
WOLFSSL_CTX* ctx;
...

wolfSSL_CTX_clear_expected_rpk(ctx);
```

See:

- wolfSSL_CTX_set_expected_rpk
- wolfSSL_clear_expected_rpk

Return:

- WOLFSSL_SUCCESS on success
- BAD_FUNC_ARG if ctx is NULL

```
int wolfSSL_clear_expected_rpk(
    WOLFSSL * ssl
)
```

Remove all pinned expected peer Raw Public Keys (RFC 7250) from the WOLFSSL object, emptying the table so it can be repopulated - for example across a peer key rotation, since the pinning APIs are otherwise append-only.

Parameters:

- **ssl** WOLFSSL object pointer *Example*

```
WOLFSSL* ssl;
...

wolfSSL_clear_expected_rpk(ssl);
```

See:

- wolfSSL_set_expected_rpk
- wolfSSL_CTX_clear_expected_rpk

Return:

- WOLFSSL_SUCCESS on success
- BAD_FUNC_ARG if ssl is NULL

```
int wolfSSL_get_scr_check_enabled(
    const WOLFSSL * ssl
)
```

Gets the state of the secure renegotiation (SCR) check requirement.

Parameters:

- **ssl** Pointer to the WOLFSSL structure, created with wolfSSL_new().

See: wolfSSL_set_scr_check_enabled

Return:

- 1 if the SCR check is enabled.
- 0 if the SCR check is disabled.
- BAD_FUNC_ARG if ssl is NULL.

This function returns whether the client requires the server to acknowledge the secure renegotiation extension and enable secure renegotiation when sending it from the client. When enabled, the client will generate a fatal handshake_failure alert if the server does not acknowledge the extension in the ServerHello message, as required by RFC 9325.

Example

```
WOLFSSL* ssl;
int enabled;

ssl = wolfSSL_new(ctx);
enabled = wolfSSL_get_scr_check_enabled(ssl);
if (enabled) {
    // SCR check is enabled
}
```

```
int wolfSSL_set_scr_check_enabled(
    WOLFSSL * ssl,
    byte enabled
)
```

Sets the state of the secure renegotiation (SCR) check requirement.

Parameters:

- **ssl** Pointer to the WOLFSSL structure, created with wolfSSL_new().
- **enabled** Non-zero to enable the SCR check, zero to disable it.

See: wolfSSL_get_scr_check_enabled

Return:

- WOLFSSL_SUCCESS on success.
- BAD_FUNC_ARG if ssl is NULL.

This function enables or disables the requirement for the server to acknowledge the secure renegotiation extension and enable secure renegotiation when sending it from the client. When enabled, the

client will generate a fatal handshake_failure alert if the server does not acknowledge the extension in the ServerHello message, as required by RFC 9325.

Example

```
WOLFSSL* ssl;
int ret;

ssl = wolfSSL_new(ctx);
ret = wolfSSL_set_scr_check_enabled(ssl, 1);
if (ret != WOLFSSL_SUCCESS) {
    // Error setting SCR check
}
```

```
int wolfSSL_CTX_DisableExtendedMasterSecret(
    WOLFSSL_CTX * ctx
)
```

Disables the TLS Extended Master Secret extension (RFC 7627) on the context: a client stops advertising it and a server ignores the peer's request, so a standard master secret is negotiated. A server also declines resumption of sessions or tickets that used EMS and does a full handshake instead. TLS 1.2 and earlier only. Requires HAVE_EXTENDED_MASTER.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

See:

- wolfSSL_DisableExtendedMasterSecret
- wolfSSL_CTX_EnableExtendedMasterSecret
- wolfSSL_CTX_RequireExtendedMasterSecret

Return:

- WOLFSSL_SUCCESS on success.
- BAD_FUNC_ARG if ctx is NULL.

Example

```
wolfSSL_CTX_DisableExtendedMasterSecret(ctx);
```

```
int wolfSSL_DisableExtendedMasterSecret(
    WOLFSSL * ssl
)
```

Disables the TLS Extended Master Secret extension (RFC 7627) on the SSL object: a client stops advertising it and a server ignores the peer's request, so a standard master secret is negotiated. A server also declines resumption of sessions or tickets that used EMS and does a full handshake instead. Call before the handshake starts, or after wolfSSL_clear. TLS 1.2 and earlier only. Requires HAVE_EXTENDED_MASTER.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_CTX_DisableExtendedMasterSecret
- wolfSSL_EnableExtendedMasterSecret
- wolfSSL_RequireExtendedMasterSecret

Return:

- WOLFSSL_SUCCESS on success.
- BAD_FUNC_ARG if ssl is NULL.
- BAD_STATE_E if the handshake has started.

Example

```
wolfSSL_DisableExtendedMasterSecret(ssl);
```

```
int wolfSSL_CTX_EnableExtendedMasterSecret(
    WOLFSSL_CTX * ctx
)
```

Re-enables the TLS Extended Master Secret extension (RFC 7627) on the context (the default): EMS is used when the peer supports it but is not mandatory. Undoes a previous disable or require. Requires HAVE_EXTENDED_MASTER.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

See:

- wolfSSL_EnableExtendedMasterSecret
- wolfSSL_CTX_DisableExtendedMasterSecret
- wolfSSL_CTX_RequireExtendedMasterSecret

Return:

- WOLFSSL_SUCCESS on success.
- BAD_FUNC_ARG if ctx is NULL.

Example

```
wolfSSL_CTX_EnableExtendedMasterSecret(ctx);
```

```
int wolfSSL_EnableExtendedMasterSecret(
    WOLFSSL * ssl
)
```

Re-enables the TLS Extended Master Secret extension (RFC 7627) on the SSL object (the default): EMS is used when the peer supports it but is not mandatory. Undoes a previous disable or require. Call before the handshake starts, or after wolfSSL_clear. Requires HAVE_EXTENDED_MASTER.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_CTX_EnableExtendedMasterSecret
- wolfSSL_DisableExtendedMasterSecret
- wolfSSL_RequireExtendedMasterSecret

Return:

- WOLFSSL_SUCCESS on success.
- BAD_FUNC_ARG if ssl is NULL.
- BAD_STATE_E if the handshake has started.

Example

```
wolfSSL_EnableExtendedMasterSecret(ssl);
```

```
int wolfSSL_CTX_RequireExtendedMasterSecret(
    WOLFSSL_CTX * ctx
)
```

Makes the TLS Extended Master Secret extension (RFC 7627) mandatory on the context: if it is not negotiated, the connection is aborted with EXT_MASTER_SECRET_NEEDED_E. A client advertises the extension even after a previous disable. TLS 1.2 and earlier only. Requires HAVE_EXTENDED_MASTER.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

See:

- wolfSSL_RequireExtendedMasterSecret
- wolfSSL_CTX_EnableExtendedMasterSecret
- wolfSSL_CTX_DisableExtendedMasterSecret

Return:

- WOLFSSL_SUCCESS on success.
- BAD_FUNC_ARG if ctx is NULL.

Example

```
wolfSSL_CTX_RequireExtendedMasterSecret(ctx);
```

```
int wolfSSL_RequireExtendedMasterSecret(
    WOLFSSL * ssl
)
```

Makes the TLS Extended Master Secret extension (RFC 7627) mandatory on the SSL object: if it is not negotiated, including on resumption, the connection is aborted with EXT_MASTER_SECRET_NEEDED_E. A client advertises the extension even after a previous disable. Call before the handshake starts, or after wolfSSL_clear. TLS 1.2 and earlier only. Requires HAVE_EXTENDED_MASTER.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_CTX_RequireExtendedMasterSecret
- wolfSSL_EnableExtendedMasterSecret
- wolfSSL_DisableExtendedMasterSecret

Return:

- WOLFSSL_SUCCESS on success.
- BAD_FUNC_ARG if ssl is NULL.
- BAD_STATE_E if the handshake has started.

Example

```
wolfSSL_RequireExtendedMasterSecret(ssl);
```

```
void * wolfSSL_GetCookieCtx(
    WOLFSSL * ssl
)
```

This function returns the IOCB_CookieCtx member of the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **ssl** SSL object

See:

- wolfSSL_SetCookieCtx
- wolfSSL_CTX_SetGenCookie
- wolfSSL_SetCookieCtx

Return:

- pointer The function returns a void pointer value stored in the IOCB_CookieCtx.
- NULL if the WOLFSSL struct is NULL
- Cookie context pointer

Gets cookie context for DTLS.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
void* cookie;
...
cookie = wolfSSL_GetCookieCtx(ssl);
if(cookie != NULL){
// You have the cookie
}
```

Example

```
WOLFSSL* ssl;
void* ctx = wolfSSL_GetCookieCtx(ssl);
```

```
int wolfSSL_SetIO_ISOTP(
    WOLFSSL * ssl,
    isotp_wolfssl_ctx * ctx,
    can_recv_fn recv_fn,
    can_send_fn send_fn,
    can_delay_fn delay_fn,
    word32 receive_delay,
    char * receive_buffer,
    int receive_buffer_size,
    void * arg
)
```

This function sets up the ISO-TP context if wolfSSL, for use when wolfSSL is compiled with WOLFSSL_ISOTP.

Parameters:

- **ssl** the wolfSSL context
- **ctx** a user created ISOTP context which this function initializes
- **recv_fn** a user CAN bus receive callback, returning the frame length, 0 if none arrived within timeout (ms), or negative on error
- **send_fn** a user CAN bus send callback
- **delay_fn** a user microsecond granularity delay function
- **receive_delay** a set amount of microseconds to delay each CAN bus packet
- **receive_buffer** a user supplied buffer to receive data, recommended that is allocated to ISOTP_DEFAULT_BUFFER_SIZE bytes
- **receive_buffer_size** - The size of receive_buffer
- **arg** an arbitrary pointer sent to recv_fn and send_fn

Return: 0 on success, WOLFSSL_CBIO_ERR_GENERAL on failure

Example

```
struct can_info can_con_info;
isotp_wolfssl_ctx isotp_ctx;
char *receive_buffer = malloc(ISOTP_DEFAULT_BUFFER_SIZE);
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(method);
WOLFSSL* ssl = wolfSSL_new(ctx);
...
wolfSSL_SetIO_ISOTP(ssl, &isotp_ctx, can_receive, can_send, can_delay, 0,
        receive_buffer, ISOTP_DEFAULT_BUFFER_SIZE, &can_con_info);
```

```
void wolfSSL_SSLDisableRead(
    WOLFSSL * ssl
)
```

This function disables reading from the IO layer.

Parameters:

- **ssl** the wolfSSL context

See:

- wolfSSL_CTX_SetIORecv
- wolfSSL_SSLSetIORecv
- wolfSSL_SSLEnableRead

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(method);
WOLFSSL* ssl = wolfSSL_new(ctx);
wolfSSL_SSLDisableRead(ssl);
```

```
void wolfSSL_SSLEnableRead(
    WOLFSSL * ssl
)
```

This function enables reading from the IO layer. Reading is enabled by default and should be used to undo wolfSSL_SSLDisableRead();.

Parameters:

- **ssl** the wolfSSL context

See:

- wolfSSL_CTX_SetIORecv
- wolfSSL_SSLSetIORecv
- wolfSSL_SSLEnableRead

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(method);
WOLFSSL* ssl = wolfSSL_new(ctx);
wolfSSL_SSLDisableRead(ssl);
...
wolfSSL_SSLEnableRead(ssl);
```

A.7 wolfSSL Error Handling and Reporting

A.6.4.147 function wolfSSL_SSLEnableRead

A.7.1 Functions

	Name
int	**wolfSSL_Debugging_ON**(void)If logging has been enabled at build time this function turns on logging at runtime. To enable logging at build time use –enable-debug or define DEBUG_WOLFSSL.
void	**wolfSSL_Debugging_OFF**(void)This function turns off runtime logging messages. If they're already off, no action is taken.

	Name
int	**wolfSSL_get_error**(WOLFSSL * ssl, int ret)This function returns a unique error code describing why the previous API function call (wolfSSL_connect, wolfSSL_accept, wolfSSL_read, wolfSSL_write, etc.) resulted in an error return code (SSL_FAILURE). The return value of the previous function is passed to wolfSSL_get_error through ret. After wolfSSL_get_error is called and returns the unique error code, wolfSSL_ERR_error_string() may be called to get a human_readable error string. See wolfSSL_ERR_error_string() for more information.
void	**wolfSSL_load_error_strings**(void)This function is for OpenSSL compatibility (SSL_load_error_string) only and takes no action.
char *	**wolfSSL_ERR_error_string**(unsigned long errNumber, char * data)This function converts an error code returned by wolfSSL_get_error() into a more human_readable error string. errNumber is the error code returned by wolfSSL_get_error() and data is the storage buffer which the error string will be placed in. The maximum length of data is 80 characters by default, as defined by MAX_ERROR_SZ is wolfssl/wolfcrypt/error.h.
void	**wolfSSL_ERR_error_string_n**(unsigned long e, char * buf, unsigned long len)This function is a version of wolfSSL_ERR_error_string() where len specifies the maximum number of characters that may be written to buf. Like wolfSSL_ERR_error_string(), this function converts an error code returned from wolfSSL_get_error() into a more human-readable error string. The human-readable string is placed in buf.
void	**wolfSSL_ERR_print_errors_fp**(XFILE fp, int err)This function converts an error code returned by wolfSSL_get_error() into a more human_readable error string and prints that string to the output file _ fp. err is the error code returned by wolfSSL_get_error() and fp is the file which the error string will be placed in.
void	**wolfSSL_ERR_print_errors_cb**(int*)(const char* str, size_t len, void *u) cb, void* u)This function uses the provided callback to handle error reporting. The callback function is executed for each error line. The string, length, and userdata are passed into the callback parameters.

	Name
int	**wolfSSL_want_read**(WOLFSSL * ssl)This function is similar to calling wolfSSL_get_error() and getting SSL_ERROR_WANT_READ in return. If the underlying error state is SSL_ERROR_WANT_READ, this function will return 1, otherwise, 0.
int	**wolfSSL_want_write**(WOLFSSL * ssl)This function is similar to calling wolfSSL_get_error() and getting SSL_ERROR_WANT_WRITE in return. If the underlying error state is SSL_ERROR_WANT_WRITE, this function will return 1, otherwise, 0.
int	**wolfSSL_want**(WOLFSSL * ssl)This function reports which I/O operation, if any, the SSL session is waiting on. It reflects the same state that wolfSSL_want_read() and wolfSSL_want_write() report individually. Unlike those two, which are always available, this function is only built when OPENSSL_EXTRA is defined.
unsigned long	**wolfSSL_ERR_peek_last_error**(void)This function returns the absolute value of the last error from WOLFSSL_ERROR encountered.

A.7.2 Functions Documentation

```
int wolfSSL_Debugging_ON(
    void
)
```

If logging has been enabled at build time this function turns on logging at runtime. To enable logging at build time use –enable-debug or define DEBUG_WOLFSSL.

Parameters:

- **none** No parameters.

See:

- wolfSSL_Debugging_OFF
- wolfSSL_SetLoggingCb

Return:

- 0 upon success.
- NOT_COMPILED_IN is the error that will be returned if logging isn't enabled for this build.

Example

```
wolfSSL_Debugging_ON();
```

```
void wolfSSL_Debugging_OFF(
    void
)
```

This function turns off runtime logging messages. If they're already off, no action is taken.

Parameters:

- **none** No parameters.

See:

- wolfSSL_Debugging_ON
- wolfSSL_SetLoggingCb

Return: none No returns.

Example

```
wolfSSL_Debugging_OFF();
```

```
int wolfSSL_get_error(
    WOLFSSL * ssl,
    int ret
)
```

This function returns a unique error code describing why the previous API function call (wolfSSL_connect, wolfSSL_accept, wolfSSL_read, wolfSSL_write, etc.) resulted in an error return code (SSL_FAILURE). The return value of the previous function is passed to wolfSSL_get_error through ret. After wolfSSL_get_error is called and returns the unique error code, wolfSSL_ERR_error_string() may be called to get a human-readable error string. See wolfSSL_ERR_error_string() for more information.

Parameters:

- **ssl** pointer to the SSL object, created with wolfSSL_new().
- **ret** return value of the previous function that resulted in an error return code.

See:

- wolfSSL_ERR_error_string
- wolfSSL_ERR_error_string_n
- wolfSSL_ERR_print_errors_fp
- wolfSSL_load_error_strings

Return:

- On successful completion, this function will return the unique error code describing why the previous API function failed.

- SSL_ERROR_NONE will be returned if ret > 0. For ret <= 0, there are some cases when this value can also be returned when a previous API appeared to return an error code but no error actually occurred. An example is calling wolfSSL_read() with a zero sz parameter. A 0 return from wolfSSL_read() usually indicates an error but in this case no error occurred. If wolfSSL_get_error() is called afterwards, SSL_ERROR_NONE will be returned.

Example

```
int err = 0;
WOLFSSL* ssl;
char buffer[80];
...
err = wolfSSL_get_error(ssl, 0);
wolfSSL_ERR_error_string(err, buffer);
printf("err = %d, %s\n", err, buffer);
```

```
void wolfSSL_load_error_strings(
    void
)
```

This function is for OpenSSL compatibility (SSL_load_error_string) only and takes no action.

Parameters:

- **none** No parameters.

See:

- wolfSSL_get_error
- wolfSSL_ERR_error_string
- wolfSSL_ERR_error_string_n
- wolfSSL_ERR_print_errors_fp
- wolfSSL_load_error_strings

Return: none No returns.

Example

```
wolfSSL_load_error_strings();
```

```
char * wolfSSL_ERR_error_string(
    unsigned long errNumber,
    char * data
)
```

This function converts an error code returned by wolfSSL_get_error() into a more human_readable error string. errNumber is the error code returned by wolfSSL_get_error() and data is the storage buffer which the error string will be placed in. The maximum length of data is 80 characters by default, as defined by MAX_ERROR_SZ is wolfssl/wolfcrypt/error.h.

Parameters:

- **errNumber** error code returned by wolfSSL_get_error().
- **data** output buffer containing human-readable error string matching errNumber.

See:

- wolfSSL_get_error
- wolfSSL_ERR_error_string_n
- wolfSSL_ERR_print_errors_fp
- wolfSSL_load_error_strings

Return:

- success On successful completion, this function returns the same string as is returned in data.
- failure Upon failure, this function returns a string with the appropriate failure reason, msg.

Example

```
int err = 0;
WOLFSSL* ssl;
char buffer[80];
...
err = wolfSSL_get_error(ssl, 0);
wolfSSL_ERR_error_string(err, buffer);
printf("err = %d, %s\n", err, buffer);
```

```
void wolfSSL_ERR_error_string_n(
    unsigned long e,
    char * buf,
    unsigned long len
)
```

This function is a version of wolfSSL_ERR_error_string() where len specifies the maximum number of characters that may be written to buf. Like wolfSSL_ERR_error_string(), this function converts an error code returned from wolfSSL_get_error() into a more human-readable error string. The human-readable string is placed in buf.

Parameters:

- **e** error code returned by wolfSSL_get_error().
- **buff** output buffer containing human-readable error string matching e.
- **len** maximum length in characters which may be written to buf.

See:

- wolfSSL_get_error
- wolfSSL_ERR_error_string
- wolfSSL_ERR_print_errors_fp

- wolfSSL_load_error_strings

Return: none No returns.

Example

```
int err = 0;
WOLFSSL* ssl;
char buffer[80];
...
err = wolfSSL_get_error(ssl, 0);
wolfSSL_ERR_error_string_n(err, buffer, 80);
printf("err = %d, %s\n", err, buffer);
```

```
void wolfSSL_ERR_print_errors_fp(
    XFILE fp,
    int err
)
```

This function converts an error code returned by wolfSSL_get_error() into a more human_readable error string and prints that string to the output file _ fp. err is the error code returned by wolfSSL_get_error() and fp is the file which the error string will be placed in.

Parameters:

- **fp** output file for human-readable error string to be written to.
- **err** error code returned by wolfSSL_get_error().

See:

- wolfSSL_get_error
- wolfSSL_ERR_error_string
- wolfSSL_ERR_error_string_n
- wolfSSL_load_error_strings

Return: none No returns.

Example

```
int err = 0;
WOLFSSL* ssl;
FILE* fp = ...
...
err = wolfSSL_get_error(ssl, 0);
wolfSSL_ERR_print_errors_fp(fp, err);
```

```
void wolfSSL_ERR_print_errors_cb(
    int(*)(const char *str, size_t len, void *u) cb,
    void * u
)
```

This function uses the provided callback to handle error reporting. The callback function is executed for each error line. The string, length, and userdata are passed into the callback parameters.

Parameters:

- **cb** the callback function.
- **u** userdata to pass into the callback function.

See:

- wolfSSL_get_error
- wolfSSL_ERR_error_string
- wolfSSL_ERR_error_string_n
- wolfSSL_load_error_strings

Return: none No returns.

Example

```
int error_cb(const char *str, size_t len, void *u)
{ fprintf((FILE*)u, "%-*.*s\n", (int)len, (int)len, str); return 0; }
...
FILE* fp = ...
wolfSSL_ERR_print_errors_cb(error_cb, fp);
```

```
int wolfSSL_want_read(
    WOLFSSL * ssl
)
```

This function is similar to calling wolfSSL_get_error() and getting SSL_ERROR_WANT_READ in return. If the underlying error state is SSL_ERROR_WANT_READ, this function will return 1, otherwise, 0.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See:

- wolfSSL_want_write
- wolfSSL_get_error

Return:

- 1 wolfSSL_get_error() would return SSL_ERROR_WANT_READ, the underlying I/O has data available for reading.
- 0 There is no SSL_ERROR_WANT_READ error state.

Example

```
int ret;
WOLFSSL* ssl = 0;
...

ret = wolfSSL_want_read(ssl);
if (ret == 1) {
    // underlying I/O has data available for reading (SSL_ERROR_WANT_READ)
}
```

```
int wolfSSL_want_write(
    WOLFSSL * ssl
)
```

This function is similar to calling wolfSSL_get_error() and getting SSL_ERROR_WANT_WRITE in return. If the underlying error state is SSL_ERROR_WANT_WRITE, this function will return 1, otherwise, 0.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See:

- wolfSSL_want_read
- wolfSSL_get_error

Return:

- 1 wolfSSL_get_error() would return SSL_ERROR_WANT_WRITE, the underlying I/O needs data to be written in order for progress to be made in the underlying SSL connection.
- 0 There is no SSL_ERROR_WANT_WRITE error state.

Example

```
int ret;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_want_write(ssl);
if (ret == 1) {
    // underlying I/O needs data to be written (SSL_ERROR_WANT_WRITE)
}
```

```
int wolfSSL_want(
    WOLFSSL * ssl
)
```

This function reports which I/O operation, if any, the SSL session is waiting on. It reflects the same state that wolfSSL_want_read() and wolfSSL_want_write() report individually. Unlike those two, which are always available, this function is only built when OPENSSL_EXTRA is defined.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See:

- wolfSSL_want_read
- wolfSSL_want_write
- wolfSSL_get_error

Return:

- WOLFSSL_READING when the underlying I/O needs data to be read before progress can be made.
- WOLFSSL_WRITING when the underlying I/O needs data to be written before progress can be made.
- WOLFSSL_NOTHING when the session is not waiting on the underlying I/O, or ssl is NULL.

Example

```
WOLFSSL* ssl = 0;
...

if (wolfSSL_want(ssl) == WOLFSSL_READING) {
    // wait for the socket to become readable, then retry
}
```

```
unsigned long wolfSSL_ERR_peek_last_error(
    void
)
```

This function returns the absolute value of the last error from WOLFSSL_ERROR encountered.

Parameters:

- **none** No parameters.

See: wolfSSL_ERR_print_errors_fp

Return: error Returns absolute value of last error.

Example

```
unsigned long err;
...
err = wolfSSL_ERR_peek_last_error();
// inspect err value
```

A.8 wolfSSL Initialization/Shutdown

A.7.2.12 function wolfSSL_ERR_peek_last_error

A.8.1 Functions

	Name
int	**wolfSSL_shutdown**(WOLFSSL * ssl)This function shuts down an active SSL/TLS connection using the SSL session, ssl. This function will try to send a “close notify” alert to the peer. The calling application can choose to wait for the peer to send its “close notify” alert in response or just go ahead and shut down the underlying connection after directly calling wolfSSL_shutdown (to save resources). Either option is allowed by the TLS specification. If the underlying connection will be used again in the future, the complete two_directional shutdown procedure must be performed to keep synchronization intact between the peers. wolfSSL_shutdown() works with both blocking and non_blocking I/O. When the underlying I/O is non_blocking, wolfSSL_shutdown() will return an error if the underlying I/O could not satisfy the needs of wolfSSL_shutdown() to continue. In this case, a call to wolfSSL_get_error() will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_shutdown() when the underlying I/O is ready.
int	**wolfSSL_SendUserCanceled**(WOLFSSL * ssl)This function sends a user_canceled alert to the peer and then shuts the connection down by calling wolfSSL_shutdown(). It is used when the application abandons a connection for its own reasons rather than because of a protocol failure.
int	**wolfSSL_SetServerID**(WOLFSSL * ssl, const unsigned char * id, int len, int newSession)This function associates the client session with the server id. If the newSession flag is on, an existing session won’t be reused.
int	**wolfSSL_library_init**(void)This function is called internally in wolfSSL_CTX_new(). This function is a wrapper around wolfSSL_Init() and exists for OpenSSL compatibility (SSL_library_init) when wolfSSL has been compiled with OpenSSL compatibility layer. wolfSSL_Init() is the more typically-used wolfSSL initialization function.

	Name
int	**wolfSSL_get_shutdown**(const WOLFSSL * ssl)This function checks the shutdown conditions in closeNotify or connReset or sentNotify members of the Options structure. The Options structure is within the WOLFSSL structure.
int	**wolfSSL_is_init_finished**(const WOLFSSL * ssl)This function checks to see if the connection is established.
int	**wolfSSL_Init**(void)Initializes the wolfSSL library for use. Must be called once per application and before any other call to the library.
int	**wolfSSL_Cleanup**(void)Un-initializes the wolfSSL library from further use. Doesn't have to be called, though it will free any resources used by the library.
int	**wolfSSL_SetMinVersion**(WOLFSSL * ssl, int version)This function sets the minimum downgrade version allowed. Applicable only when the connection allows downgrade using (wolfSSLv23_client_method or wolfSSLv23_server_method).
int	**wolfSSL_ALPN_GetProtocol**(WOLFSSL * ssl, char ** protocol_name, unsigned short * size)This function gets the protocol name set by the server.
int	**wolfSSL_ALPN_GetPeerProtocol**(WOLFSSL * ssl, char ** list, unsigned short * listSz)This function copies the alpn_client_list data from the SSL object to the buffer.
int	**wolfSSL_MakeTlsMasterSecret**(unsigned char * ms, word32 msLen, const unsigned char * pms, word32 pmsLen, const unsigned char * cr, const unsigned char * sr, int tls1_2, int hash_type)This function copies the values of cr and sr then passes through to wc_PRF (pseudo random function) and returns that value.
int	**wolfSSL_preferred_group**(WOLFSSL * ssl)This function returns the key exchange group the client prefers to use in the TLS v1.3 handshake. Call this function to after a handshake is complete to determine which group the server prefers so that this information can be used in future connections to pre-generate a key pair for key exchange.
void	**wolfSSL_CTX_set_client_CA_list**(WOLFSSL_CTX * ctx, WOLF_STACK_OF(WOLFSSL_X509_NAME) * names)On the server, this sets a list of CA names to be sent to clients in certificate requests as a hint for which CA's are supported by the server.

	Name
WOLFSSL_STACK *	**wolfSSL_CTX_get_client_CA_list**(const WOLFSSL_CTX * ctx)This retrieves the list previously set via wolfSSL_CTX_set_client_CA_list, or NULL if no list has been set.
void	**wolfSSL_set_client_CA_list**(WOLFSSL * ssl, WOLF_STACK_OF(WOLFSSL_X509_NAME) * names)Same as wolfSSL_CTX_set_client_CA_list, but specific to a session. If a CA list is set on both the context and the session, the list on the session is used.
WOLFSSL_STACK *	**wolfSSL_get_client_CA_list**(const WOLFSSL * ssl)On the server, this retrieves the list previously set via wolfSSL_set_client_CA_list. If none was set, returns the list previously set via wolfSSL_CTX_set_client_CA_list. If no list at all was set, returns NULL.
void	**wolfSSL_CTX_set0_CA_list**(WOLFSSL_CTX * ctx, WOLF_STACK_OF(WOLFSSL_X509_NAME) * names)This function sets a list of CA names to be sent to the peer as a hint for which CA's are supported for its authentication.
WOLFSSL_STACK *	**wolfSSL_CTX_get0_CA_list**(const WOLFSSL_CTX * ctx)This retrieves the list previously set via wolfSSL_CTX_set0_CA_list, or NULL if no list has been set.
void	**wolfSSL_set0_CA_list**(WOLFSSL * ssl, WOLF_STACK_OF(WOLFSSL_X509_NAME) * names)Same as wolfSSL_CTX_set0_CA_list, but specific to a session. If a CA list is set on both the context and the session, the list on the session is used.
WOLFSSL_STACK *	**wolfSSL_get0_CA_list**(const WOLFSSL * ssl)This retrieves the list previously set via wolfSSL_set0_CA_list. If none was set, returns the list previously set via wolfSSL_CTX_set0_CA_list. If no list at all was set, returns NULL.
WOLFSSL_STACK *	**wolfSSL_get0_peer_CA_list**(const WOLFSSL * ssl)This returns the CA list received from the peer.
int	**wolfSSL_get_client_suites_sigalgs**(const WOLFSSL * ssl, const byte ** suites, word16 * suiteSz, const byte ** hashSigAlgo, word16 * hashSigAlgoSz)This function returns the raw list of ciphersuites and signature algorithms offered by the client. The lists are only stored and returned inside a callback setup with wolfSSL_CTX_set_cert_cb(). This is useful to be able to dynamically load certificates and keys based on the available ciphersuites and signature algorithms.

	Name
WOLFSSL_CIPHERSUITE_INFO	**wolfSSL_get_ciphersuite_info**(byte first, byte second)This returns information about the ciphersuite directly from the raw ciphersuite bytes.
int	**wolfSSL_get_sigalg_info**(byte first, byte second, int * hashAlgo, int * sigAlgo)This returns information about the hash and signature algorithm directly from the raw ciphersuite bytes.
int	**wolfSSL_UseCertificateAuthority**(WOLFSSL * ssl, const unsigned char * dn, unsigned int dnSz)Adds a CA distinguished name to the list of certificate authorities announced via the TLS 1.3 certificate_authorities extension (RFC 8446 section 4.2.4) on the given SSL session. The DN must be the inner content of a DER_encoded X.509 Name (the bytes after the SEQUENCE tag and length), as returned by wc_GetDecodedCertSubjectRaw(). The library copies the bytes and prepends the SEQUENCE header on the wire automatically.
int	**wolfSSL_CTX_UseCertificateAuthority**(WOLFSSL_CTX * ctx, const unsigned char * dn, unsigned int dnSz)Adds a CA distinguished name to the list of certificate authorities announced via the TLS 1.3 certificate_authorities extension (RFC 8446 section 4.2.4) on all SSL sessions created from this context. The DN format and requirements are identical to wolfSSL_UseCertificateAuthority().
void	**wolfSSL_ClearCertificateAuthorities**(WOLFSSL * ssl)Frees and removes all CA distinguished names previously added to the SSL session via wolfSSL_UseCertificateAuthority(). After this call the session_level native CA list is empty; the CTX_level list (if any) is not affected.
void	**wolfSSL_CTX_ClearCertificateAuthorities**(WOLFSSL_CTX * ctx)Frees and removes all CA distinguished names previously added to the context via wolfSSL_CTX_UseCertificateAuthority(). After this call the CTX-level native CA list is empty.
int	**wolfSSL_GetPeerCertificateAuthorityCount**(const WOLFSSL * ssl)Returns the number of CA distinguished names received from the peer in the TLS 1.3 certificate_authorities extension. This is typically called inside a cert_cb (WOLFSSL_CERT_SETUP_CB) on the server side to inspect which CAs the client trusts.

	Name
int	**wolfSSL_GetPeerCertificateAuthority**(const WOLFSSL * ssl, int idx, unsigned char * outDn, unsigned int outDnSz)Copies the idx_th CA distinguished name received from the peer in the TLS 1.3 certificate_authorities extension into the caller's buffer. The DN is the inner content of the DER_encoded Name (without the SEQUENCE header), matching the format accepted by wolfSSL_UseCertificateAuthority().

A.8.2 Functions Documentation

```
int wolfSSL_shutdown(
    WOLFSSL * ssl
)
```

This function shuts down an active SSL/TLS connection using the SSL session, ssl. This function will try to send a "close notify" alert to the peer. The calling application can choose to wait for the peer to send its "close notify" alert in response or just go ahead and shut down the underlying connection after directly calling wolfSSL_shutdown (to save resources). Either option is allowed by the TLS specification. If the underlying connection will be used again in the future, the complete two-directional shutdown procedure must be performed to keep synchronization intact between the peers. wolfSSL_shutdown() works with both blocking and non-blocking I/O. When the underlying I/O is non-blocking, wolfSSL_shutdown() will return an error if the underlying I/O could not satisfy the needs of wolfSSL_shutdown() to continue. In this case, a call to wolfSSL_get_error() will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_shutdown() when the underlying I/O is ready.

Parameters:

- **ssl** pointer to the SSL session created with wolfSSL_new().

See:

- wolfSSL_free
- wolfSSL_CTX_free

Return:

- SSL_SUCCESS will be returned upon success.
- SSL_SHUTDOWN_NOT_DONE will be returned when shutdown has not finished, and the function should be called again.
- SSL_FATAL_ERROR will be returned upon failure. Call wolfSSL_get_error() for a more specific error code.

When the connection is already closed or reset and no close notify was ever sent, the exchange can never complete and SSL_FATAL_ERROR is returned, so a loop that calls this function until the shutdown completes terminates. wolfSSL_get_error() then reports SOCKET_PEER_CLOSED_E, unless a more specific error has already been recorded. That code is used whatever closed the connection, including this side sending a fatal alert, so under OPENSSL_EXTRA, where wolfSSL_get_error() reports it as SSL_ERROR_SYSCALL, a locally aborted connection can surface as a syscall error. Recording it also puts

an entry on the OpenSSL error queue in builds that have one, so an application that inspects the queue after tearing down an already_aborted connection finds an entry where it previously found none; call wolfSSL_ERR_clear_error() if leftover entries matter.

Example

```
#include <wolfssl/ssl.h>

int ret = 0;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_shutdown(ssl);
if (ret != 0) {
    // failed to shut down SSL connection
}
```

```
int wolfSSL_SendUserCanceled(
    WOLFSSL * ssl
)
```

This function sends a user_canceled alert to the peer and then shuts the connection down by calling wolfSSL_shutdown(). It is used when the application abandons a connection for its own reasons rather than because of a protocol failure.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See:

- wolfSSL_shutdown
- wolfSSL_get_error

Return:

- WOLFSSL_SUCCESS on successful shutdown.
- WOLFSSL_SHUTDOWN_NOT_DONE when the peer has yet to send its close notify alert. Call wolfSSL_shutdown() again to complete the bidirectional shutdown. Under WOLFSSL_ERROR_CODE_OPENSSL this value is 0, which is also WOLFSSL_FAILURE, so in that configuration the return value alone does not separate this case from the one below.
- WOLFSSL_FAILURE when ssl is NULL or the alert could not be sent. Call wolfSSL_get_error() for the reason.
- WOLFSSL_FATAL_ERROR when the shutdown that follows the alert fails. Call wolfSSL_get_error() for the reason.
- SSL_SHUTDOWN_ALREADY_DONE_E when the connection was already shut down and WOLFSSL_SHUTDOWNONCE is defined.

Note: RFC 9846, Section 6.1 requires this alert to be followed by a close notify, which is why the shutdown is part of this call. On the receiving side the alert is not itself an error: a TLS 1.3 peer keeps reading until the close notify arrives, whatever AlertLevel was used.

Example

```
int ret = 0;
WOLFSSL* ssl = 0;
...

ret = wolfSSL_SendUserCanceled(ssl);
if (ret != WOLFSSL_SUCCESS) {
    // failed to shut the connection down, see wolfSSL_get_error()
}
```

```
int wolfSSL_SetServerID(
    WOLFSSL * ssl,
    const unsigned char * id,
    int len,
    int newSession
)
```

This function associates the client session with the server id. If the newSession flag is on, an existing session won't be reused.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **id** a constant byte pointer that will be copied to the serverID member of the WOLFSSL_SESSION structure.
- **len** an int type representing the length of the session id parameter.
- **newSession** an int type representing the flag to denote whether to reuse a session or not.

See: wolfSSL_set_session

Return:

- SSL_SUCCESS returned if the function executed without error.
- BAD_FUNC_ARG returned if the WOLFSSL struct or id parameter is NULL or if len is not greater than zero.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol );
WOLFSSL* ssl = WOLFSSL_new(ctx);
const byte id[MAX_SIZE];  // or dynamically create space
int len = 0; // initialize length
int newSession = 0; // flag to allow
…
int ret = wolfSSL_SetServerID(ssl, id, len, newSession);

if (ret == WOLFSSL_SUCCESS) {
    // The Id was successfully set
}
```

```
int wolfSSL_library_init(
    void
)
```

This function is called internally in wolfSSL_CTX_new(). This function is a wrapper around wolfSSL_Init() and exists for OpenSSL compatibility (SSL_library_init) when wolfSSL has been compiled with OpenSSL compatibility layer. wolfSSL_Init() is the more typically-used wolfSSL initialization function.

Parameters:

- **none** No parameters.

See:

- wolfSSL_Init
- wolfSSL_Cleanup

Return:

- SSL_SUCCESS If successful the call will return.
- SSL_FATAL_ERROR is returned upon failure.

Example

```
int ret = 0;
ret = wolfSSL_library_init();
if (ret != SSL_SUCCESS) {
    failed to initialize wolfSSL
}
...
```

```
int wolfSSL_get_shutdown(
    const WOLFSSL * ssl
)
```

This function checks the shutdown conditions in closeNotify or connReset or sentNotify members of the Options structure. The Options structure is within the WOLFSSL structure.

Parameters:

- **ssl** a constant pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_SESSION_free

Return:

- 1 SSL_SENT_SHUTDOWN is returned.

- 2 SSL_RECEIVED_SHUTDOWN is returned.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = WOLFSSL_CTX_new( protocol method );
WOLFSSL* ssl = WOLFSSL_new(ctx);
…
int ret;
ret = wolfSSL_get_shutdown(ssl);

if(ret == 1){
    SSL_SENT_SHUTDOWN
} else if(ret == 2){
    SSL_RECEIVED_SHUTDOWN
} else {
    Fatal error.
}
```

```
int wolfSSL_is_init_finished(
    const WOLFSSL * ssl
)
```

This function checks to see if the connection is established.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_set_accept_state
- wolfSSL_get_keys
- wolfSSL_set_shutdown

Return:

- 0 returned if the connection is not established, i.e. the WOLFSSL struct is NULL or the handshake is not done.
- 1 returned if the connection is established i.e. the WOLFSSL handshake is done.

EXAMPLE

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_is_init_finished(ssl)){
```

```
    Handshake is done and connection is established
}
```

```
int wolfSSL_Init(
    void
)
```

Initializes the wolfSSL library for use. Must be called once per application and before any other call to the library.

See: wolfSSL_Cleanup

Return:

- SSL_SUCCESS If successful the call will return.
- BAD_MUTEX_E is an error that may be returned.
- WC_INIT_E wolfCrypt initialization error returned.

Example

```
int ret = 0;
ret = wolfSSL_Init();
if (ret != SSL_SUCCESS) {
    failed to initialize wolfSSL library
}
```

```
int wolfSSL_Cleanup(
    void
)
```

Un-initializes the wolfSSL library from further use. Doesn't have to be called, though it will free any resources used by the library.

See: wolfSSL_Init

Return:

- SSL_SUCCESS return no errors.
- BAD_MUTEX_E a mutex error return.]

Example

```
wolfSSL_Cleanup();
```

```
int wolfSSL_SetMinVersion(
    WOLFSSL * ssl,
    int version
)
```

This function sets the minimum downgrade version allowed. Applicable only when the connection allows downgrade using (wolfSSLv23_client_method or wolfSSLv23_server_method).

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **version** an integer representation of the version to be set as the minimum: WOLFSSL_SSLV3 = 0, WOLFSSL_TLSV1 = 1, WOLFSSL_TLSV1_1 = 2 or WOLFSSL_TLSV1_2 = 3.

See: SetMinVersionHelper

Return:

- SSL_SUCCESS returned if this function and its subroutine executes without error.
- BAD_FUNC_ARG returned if the SSL object is NULL. In the subroutine this error is thrown if there is not a good version match.

Example

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(protocol method);
WOLFSSL* ssl = WOLFSSL_new(ctx);
int version;  macro representation
…
if(wolfSSL_CTX_SetMinVersion(ssl->ctx, version) != SSL_SUCCESS){
    Failed to set min version
}
```

```
int wolfSSL_ALPN_GetProtocol(
    WOLFSSL * ssl,
    char ** protocol_name,
    unsigned short * size
)
```

This function gets the protocol name set by the server.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **protocol_name** a pointer to a char that represents the protocol name and will be held in the ALPN structure.
- **size** a word16 type that represents the size of the protocol_name.

See:

- TLSX_ALPN_GetRequest

- TLSX_Find

Return:

- SSL_SUCCESS returned on successful execution where no errors were thrown.
- SSL_FATAL_ERROR returned if the extension was not found or if there was no protocol match with peer. There will also be an error thrown if there is more than one protocol name accepted.
- SSL_ALPN_NOT_FOUND returned signifying that no protocol match with peer was found.
- BAD_FUNC_ARG returned if there was a NULL argument passed into the function.

Example

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new( protocol method );
WOLFSSL* ssl = WOLFSSL_new(ctx);
...
int err;
char* protocol_name = NULL;
Word16 protocol_nameSz = 0;
err = wolfSSL_ALPN_GetProtocol(ssl, &protocol_name, &protocol_nameSz);

if(err == SSL_SUCCESS){
    // Sent ALPN protocol
}
```

```
int wolfSSL_ALPN_GetPeerProtocol(
    WOLFSSL * ssl,
    char ** list,
    unsigned short * listSz
)
```

This function copies the alpn_client_list data from the SSL object to the buffer.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **list** a pointer to the buffer. The data from the SSL object will be copied into it.
- **listSz** the buffer size.

See: wolfSSL_UseALPN

Return:

- SSL_SUCCESS returned if the function executed without error. The alpn_client_list member of the SSL object has been copied to the list parameter.
- BAD_FUNC_ARG returned if the list or listSz parameter is NULL.
- BUFFER_ERROR returned if there will be a problem with the list buffer (either it's NULL or the size is 0).
- MEMORY_ERROR returned if there was a problem dynamically allocating memory.

Example

```
#import <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method);
WOLFSSL* ssl = wolfSSL_new(ctx);
…
#ifdef HAVE_ALPN
char* list = NULL;
word16 listSz = 0;
…
err = wolfSSL_ALPN_GetPeerProtocol(ssl, &list, &listSz);

if(err == SSL_SUCCESS){
    List of protocols names sent by client
}



int wolfSSL_MakeTlsMasterSecret(
    unsigned char * ms,
    word32 msLen,
    const unsigned char * pms,
    word32 pmsLen,
    const unsigned char * cr,
    const unsigned char * sr,
    int tls1_2,
    int hash_type
)
```

This function copies the values of cr and sr then passes through to wc_PRF (pseudo random function) and returns that value.

Parameters:

- **ms** the master secret held in the Arrays structure.
- **msLen** the length of the master secret.
- **pms** the pre-master secret held in the Arrays structure.
- **pmsLen** the length of the pre-master secret.
- **cr** the client random.
- **sr** the server random.
- **tls1_2** signifies that the version is at least tls version 1.2.
- **hash_type** signifies the hash type.

See:

- wc_PRF
- MakeTlsMasterSecret

Return:

- 0 on success
- BUFFER_E returned if there will be an error with the size of the buffer.
- MEMORY_E returned if a subroutine failed to allocate dynamic memory.

Example

```
WOLFSSL* ssl;

called in MakeTlsMasterSecret and retrieves the necessary
information as follows:

int MakeTlsMasterSecret(WOLFSSL* ssl){
int ret;
ret = wolfSSL_makeTlsMasterSecret(ssl->arrays->masterSecret, SECRET_LEN,
ssl->arrays->preMasterSecret, ssl->arrays->preMasterSz,
ssl->arrays->clientRandom, ssl->arrays->serverRandom,
IsAtLeastTLSv1_2(ssl), ssl->specs.mac_algorithm);
…
return ret;

}
```

```
int wolfSSL_preferred_group(
    WOLFSSL * ssl
)
```

This function returns the key exchange group the client prefers to use in the TLS v1.3 handshake. Call this function to after a handshake is complete to determine which group the server prefers so that this information can be used in future connections to pre-generate a key pair for key exchange.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_UseKeyShare
- wolfSSL_CTX_set_groups
- wolfSSL_set_groups
- wolfSSL_CTX_set1_groups_list
- wolfSSL_set1_groups_list

Return:

- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3.
- SIDE_ERROR if called with a server.
- NOT_READY_ERROR if called before handshake is complete.
- Group identifier if successful.

Example

```
int ret;
int group;
```

```
WOLFSSL* ssl;
...
ret = wolfSSL_CTX_set1_groups_list(ssl)
if (ret < 0) {
    // failed to get group
}
group = ret;
```

```
void wolfSSL_CTX_set_client_CA_list(
    WOLFSSL_CTX * ctx,
    WOLF_STACK_OF(WOLFSSL_X509_NAME) * names
)
```

On the server, this sets a list of CA names to be sent to clients in certificate requests as a hint for which CA's are supported by the server.

Parameters:

- **ctx** Pointer to the wolfSSL context
- **names** List of names to be set

See:

- wolfSSL_set_client_CA_list
- wolfSSL_CTX_get_client_CA_list
- wolfSSL_get_client_CA_list
- wolfSSL_CTX_set0_CA_list
- wolfSSL_set0_CA_list
- wolfSSL_CTX_get0_CA_list
- wolfSSL_get0_CA_list
- wolfSSL_get0_peer_CA_list

On the client, this function has no effect.

```
WOLFSSL_STACK * wolfSSL_CTX_get_client_CA_list(
    const WOLFSSL_CTX * ctx
)
```

This retrieves the list previously set via wolfSSL_CTX_set_client_CA_list, or NULL if no list has been set.

Parameters:

- **ctx** Pointer to the wolfSSL context

See:

- wolfSSL_set_client_CA_list
- wolfSSL_CTX_set_client_CA_list
- wolfSSL_get_client_CA_list

- wolfSSL_CTX_set0_CA_list
- wolfSSL_set0_CA_list
- wolfSSL_CTX_get0_CA_list
- wolfSSL_get0_CA_list
- wolfSSL_get0_peer_CA_list

Return: A stack of WOLFSSL_X509_NAMEs containing the CA names

```
void wolfSSL_set_client_CA_list(
    WOLFSSL * ssl,
    WOLF_STACK_OF(WOLFSSL_X509_NAME) * names
)
```

Same as wolfSSL_CTX_set_client_CA_list, but specific to a session. If a CA list is set on both the context and the session, the list on the session is used.

Parameters:

- **ssl** Pointer to the WOLFSSL object
- **names** List of names to be set.

See:

- wolfSSL_CTX_set_client_CA_list
- wolfSSL_CTX_get_client_CA_list
- wolfSSL_get_client_CA_list
- wolfSSL_CTX_set0_CA_list
- wolfSSL_set0_CA_list
- wolfSSL_CTX_get0_CA_list
- wolfSSL_get0_CA_list
- wolfSSL_get0_peer_CA_list

```
WOLFSSL_STACK * wolfSSL_get_client_CA_list(
    const WOLFSSL * ssl
)
```

On the server, this retrieves the list previously set via wolfSSL_set_client_CA_list. If none was set, returns the list previously set via wolfSSL_CTX_set_client_CA_list. If no list at all was set, returns NULL.

Parameters:

- **ssl** Pointer to the WOLFSSL object

See:

- wolfSSL_CTX_set_cert_cb
- wolfSSL_CTX_set_client_CA_list
- wolfSSL_CTX_get_client_CA_list
- wolfSSL_get_client_CA_list

- wolfSSL_CTX_set0_CA_list
- wolfSSL_set0_CA_list
- wolfSSL_CTX_get0_CA_list
- wolfSSL_get0_CA_list
- wolfSSL_get0_peer_CA_list

Return: A stack of WOLFSSL_X509_NAMEs containing the CA names

On the client, this retrieves the list that was received from the server, or NULL if none was received. wolfSSL_CTX_set_cert_cb can be used to register a callback to dynamically load certificates when a certificate request is received from the server.

```
void wolfSSL_CTX_set0_CA_list(
    WOLFSSL_CTX * ctx,
    WOLF_STACK_OF(WOLFSSL_X509_NAME) * names
)
```

This function sets a list of CA names to be sent to the peer as a hint for which CA's are supported for its authentication.

Parameters:

- **ctx** Pointer to the wolfSSL context
- **names** List of names to be set

See:

- wolfSSL_CTX_set_client_CA_list
- wolfSSL_set_client_CA_list
- wolfSSL_CTX_get_client_CA_list
- wolfSSL_get_client_CA_list
- wolfSSL_set0_CA_list
- wolfSSL_CTX_get0_CA_list
- wolfSSL_get0_CA_list
- wolfSSL_get0_peer_CA_list

In TLS >= 1.3, this is supported in both directions between the client and the server. On the server, the CA names will be sent as part of a CertificateRequest, making this function an equivalent of *_set_client_CA_list; on the client, these are sent as part of ClientHello.

In TLS < 1.3, sending CA names from the client to the server is not supported, therefore this function is equivalent to wolfSSL_CTX_set_client_CA_list.

Note that the lists set via *_set_client_CA_list and *_set0_CA_list are separate internally, i.e. calling *_get_client_CA_list will not retrieve a list set via *_set0_CA_list and vice versa. If both are set, the server will ignore *_set0_CA_list when sending CA names to the client.

```
WOLFSSL_STACK * wolfSSL_CTX_get0_CA_list(
    const WOLFSSL_CTX * ctx
)
```

This retrieves the list previously set via wolfSSL_CTX_set0_CA_list, or NULL if no list has been set.

Parameters:

- **ctx** Pointer to the wolfSSL context

See:

- wolfSSL_CTX_set_client_CA_list
- wolfSSL_set_client_CA_list
- wolfSSL_CTX_get_client_CA_list
- wolfSSL_get_client_CA_list
- wolfSSL_CTX_set0_CA_list
- wolfSSL_set0_CA_list
- wolfSSL_get0_CA_list
- wolfSSL_get0_peer_CA_list

Return: A stack of WOLFSSL_X509_NAMEs containing the CA names

```
void wolfSSL_set0_CA_list(
    WOLFSSL * ssl,
    WOLF_STACK_OF(WOLFSSL_X509_NAME) * names
)
```

Same as wolfSSL_CTX_set0_CA_list, but specific to a session. If a CA list is set on both the context and the session, the list on the session is used.

Parameters:

- **ssl** Pointer to the WOLFSSL object
- **names** List of names to be set.

See:

- wolfSSL_CTX_set_client_CA_list
- wolfSSL_set_client_CA_list
- wolfSSL_CTX_get_client_CA_list
- wolfSSL_get_client_CA_list
- wolfSSL_CTX_set0_CA_list
- wolfSSL_CTX_get0_CA_list
- wolfSSL_get0_CA_list
- wolfSSL_get0_peer_CA_list

```
WOLFSSL_STACK * wolfSSL_get0_CA_list(
    const WOLFSSL * ssl
)
```

This retrieves the list previously set via wolfSSL_set0_CA_list. If none was set, returns the list previously set via wolfSSL_CTX_set0_CA_list. If no list at all was set, returns NULL.

Parameters:

- **ssl** Pointer to the WOLFSSL object

See:

- wolfSSL_CTX_set_client_CA_list
- wolfSSL_set_client_CA_list
- wolfSSL_CTX_get_client_CA_list
- wolfSSL_get_client_CA_list
- wolfSSL_CTX_set0_CA_list
- wolfSSL_set0_CA_list
- wolfSSL_CTX_get0_CA_list
- wolfSSL_get0_peer_CA_list

Return: A stack of WOLFSSL_X509_NAMEs containing the CA names

```
WOLFSSL_STACK * wolfSSL_get0_peer_CA_list(
    const WOLFSSL * ssl
)
```

This returns the CA list received from the peer.

Parameters:

- **ssl** Pointer to the WOLFSSL object

See:

- wolfSSL_CTX_set_cert_cb
- wolfSSL_CTX_set_client_CA_list
- wolfSSL_set_client_CA_list
- wolfSSL_CTX_get_client_CA_list
- wolfSSL_get_client_CA_list
- wolfSSL_CTX_set0_CA_list
- wolfSSL_set0_CA_list
- wolfSSL_CTX_get0_CA_list
- wolfSSL_get0_CA_list

Return: A stack of WOLFSSL_X509_NAMEs containing the CA names

On the client, this is the list sent by the server in a CertificateRequest, and this function is equivalent to wolfSSL_get_client_CA_list.

On the server, this is the list sent by the client in the ClientHello message in TLS >= 1.3; in TLS < 1.3, the function always returns NULL on the server side.

wolfSSL_CTX_set_cert_cb can be used to register a callback to dynamically load certificates when a CA list is received from the peer.

```
int wolfSSL_get_client_suites_sigalgs(
    const WOLFSSL * ssl,
    const byte ** suites,
    word16 * suiteSz,
    const byte ** hashSigAlgo,
    word16 * hashSigAlgoSz
)
```

This function returns the raw list of ciphersuites and signature algorithms offered by the client. The lists are only stored and returned inside a callback setup with wolfSSL_CTX_set_cert_cb(). This is useful to be able to dynamically load certificates and keys based on the available ciphersuites and signature algorithms.

Parameters:

- **ssl** The WOLFSSL object to extract the lists from.
- **suites** Raw and unfiltered list of client ciphersuites. May be NULL if no suites are available.
- **suiteSz** Size of suites in bytes.
- **hashSigAlgo** Raw and unfiltered list of client signature algorithms. May be NULL if not provided.
- **hashSigAlgoSz** Size of hashSigAlgo in bytes.

See:

- wolfSSL_get_ciphersuite_info
- wolfSSL_get_sigalg_info

Return:

- WOLFSSL_SUCCESS when suites available
- WOLFSSL_FAILURE when suites not available

Example

```
int certCB(WOLFSSL* ssl, void* arg)
{
    const byte* suites = NULL;
    word16 suiteSz = 0;
    const byte* hashSigAlgo = NULL;
    word16 hashSigAlgoSz = 0;

    wolfSSL_get_client_suites_sigalgs(ssl, &suites, &suiteSz, &hashSigAlgo,
            &hashSigAlgoSz);

    // Choose certificate to load based on ciphersuites and sigalgs
}

WOLFSSL* ctx;
ctx  = wolfSSL_CTX_new(wolfTLSv1_3_method_ex(NULL));
wolfSSL_CTX_set_cert_cb(ctx, certCB, NULL);
```

```
WOLFSSL_CIPHERSUITE_INFO wolfSSL_get_ciphersuite_info(
    byte first,
    byte second
)
```

This returns information about the ciphersuite directly from the raw ciphersuite bytes.

Parameters:

- **first** First byte of the ciphersuite
- **second** Second byte of the ciphersuite

See:

- wolfSSL_get_client_suites_sigalgs
- wolfSSL_get_sigalg_info

Return: WOLFSSL_CIPHERSUITE_INFO A struct containing information about the type of authentication used in the ciphersuite.

Example

```
WOLFSSL_CIPHERSUITE_INFO info =
        wolfSSL_get_ciphersuite_info(suites[0], suites[1]);
if (info.rsaAuth)
    haveRSA = 1;
else if (info.eccAuth)
    haveECC = 1;
```

```
int wolfSSL_get_sigalg_info(
    byte first,
    byte second,
    int * hashAlgo,
    int * sigAlgo
)
```

This returns information about the hash and signature algorithm directly from the raw ciphersuite bytes.

Parameters:

- **first** First byte of the hash and signature algorithm
- **second** Second byte of the hash and signature algorithm
- **hashAlgo** The enum wc_HashType of the MAC algorithm
- **sigAlgo** The enum Key_Sum of the authentication algorithm

See:

- wolfSSL_get_client_suites_sigalgs
- wolfSSL_get_ciphersuite_info

Return:

- 0 when info was correctly set
- BAD_FUNC_ARG when either input parameters are NULL or the bytes are not a recognized sigalg suite

Example

```
enum wc_HashType hashAlgo;
enum Key_Sum sigAlgo;

wolfSSL_get_sigalg_info(hashSigAlgo[idx+0], hashSigAlgo[idx+1],
        &hashAlgo, &sigAlgo);

if (sigAlgo == RSAk || sigAlgo == RSAPSSk)
    haveRSA = 1;
else if (sigAlgo == ECDSAk)
    haveECC = 1;
```

```
int wolfSSL_UseCertificateAuthority(
    WOLFSSL * ssl,
    const unsigned char * dn,
    unsigned int dnSz
)
```

Adds a CA distinguished name to the list of certificate authorities announced via the TLS 1.3 certificate_authorities extension (RFC 8446 section 4.2.4) on the given SSL session. The DN must be the inner content of a DER-encoded X.509 Name (the bytes after the SEQUENCE tag and length), as returned by wc_GetDecodedCertSubjectRaw(). The library copies the bytes and prepends the SEQUENCE header on the wire automatically.

Parameters:

- **ssl** pointer to a WOLFSSL object, created with wolfSSL_new().
- **dn** pointer to the DER-encoded subject Name content.
- **dnSz** size in bytes of the DN content.

See:

- wolfSSL_CTX_UseCertificateAuthority
- wolfSSL_ClearCertificateAuthorities
- wolfSSL_GetPeerCertificateAuthorityCount
- wolfSSL_GetPeerCertificateAuthority
- wc_GetDecodedCertSubjectRaw

Return:

- 0 on success.
- BAD_FUNC_ARG if ssl or dn is NULL, dnSz is 0, or dnSz exceeds the maximum content size.

- MEMORY_ERROR if memory allocation fails.

Multiple DNs may be added; each call appends to the list. Use wolfSSL_ClearCertificateAuthorities() to reset the list.

Requires WOLFSSL_TLS13 and !NO_CERTS and !WOLFSSL_NO_CA_NAMES.

Example

```
DecodedCert decoded;
const byte* subject = NULL;
int subjectSz = 0;

wc_InitDecodedCert(&decoded, certDer, certDerSz, NULL);
wc_ParseCert(&decoded, CERT_TYPE, NO_VERIFY, NULL);
wc_GetDecodedCertSubjectRaw(&decoded, &subject, &subjectSz);

ret = wolfSSL_UseCertificateAuthority(ssl, subject,
        (unsigned int)subjectSz);
if (ret != 0) {
    // error adding CA DN
}
wc_FreeDecodedCert(&decoded);
```

```
int wolfSSL_CTX_UseCertificateAuthority(
    WOLFSSL_CTX * ctx,
    const unsigned char * dn,
    unsigned int dnSz
)
```

Adds a CA distinguished name to the list of certificate authorities announced via the TLS 1.3 certificate_authorities extension (RFC 8446 section 4.2.4) on all SSL sessions created from this context. The DN format and requirements are identical to wolfSSL_UseCertificateAuthority().

Parameters:

- **ctx** pointer to a WOLFSSL_CTX object, created with wolfSSL_CTX_new().
- **dn** pointer to the DER-encoded subject Name content.
- **dnSz** size in bytes of the DN content.

See:

- wolfSSL_UseCertificateAuthority
- wolfSSL_CTX_ClearCertificateAuthorities
- wc_GetDecodedCertSubjectRaw

Return:

- 0 on success.
- BAD_FUNC_ARG if ctx or dn is NULL, dnSz is 0, or dnSz exceeds the maximum content size.
- MEMORY_ERROR if memory allocation fails.

Per-session lists set via wolfSSL_UseCertificateAuthority() take precedence; if the SSL object has its own list, the CTX list is not sent.

Requires WOLFSSL_TLS13 and !NO_CERTS and !WOLFSSL_NO_CA_NAMES.

Example

```
ret = wolfSSL_CTX_UseCertificateAuthority(ctx, subject,
        (unsigned int)subjectSz);
if (ret != 0) {
    // error adding CA DN
}
```

```
void wolfSSL_ClearCertificateAuthorities(
    WOLFSSL * ssl
)
```

Frees and removes all CA distinguished names previously added to the SSL session via wolfSSL_UseCertificateAuthority(). After this call the session_level native CA list is empty; the CTX_level list (if any) is not affected.

Parameters:

- **ssl** pointer to a WOLFSSL object, created with wolfSSL_new().

See:

- wolfSSL_UseCertificateAuthority
- wolfSSL_CTX_ClearCertificateAuthorities

Return: none No return value.

Requires WOLFSSL_TLS13 and !NO_CERTS and !WOLFSSL_NO_CA_NAMES.

Example

```
wolfSSL_UseCertificateAuthority(ssl, dn1, dn1Sz);
wolfSSL_UseCertificateAuthority(ssl, dn2, dn2Sz);
// Clear all session-level CA DNs:
wolfSSL_ClearCertificateAuthorities(ssl);
```

```
void wolfSSL_CTX_ClearCertificateAuthorities(
    WOLFSSL_CTX * ctx
)
```

Frees and removes all CA distinguished names previously added to the context via wolfSSL_CTX_UseCertificateAuthorit
After this call the CTX-level native CA list is empty.

Parameters:

- **ctx** pointer to a WOLFSSL_CTX object, created with wolfSSL_CTX_new().

See:

- wolfSSL_CTX_UseCertificateAuthority
- wolfSSL_ClearCertificateAuthorities

Return: none No return value.

Requires WOLFSSL_TLS13 and !NO_CERTS and !WOLFSSL_NO_CA_NAMES.

Example

```
wolfSSL_CTX_UseCertificateAuthority(ctx, dn, dnSz);
// Clear all CTX-level CA DNs:
wolfSSL_CTX_ClearCertificateAuthorities(ctx);
```

```
int wolfSSL_GetPeerCertificateAuthorityCount(
    const WOLFSSL * ssl
)
```

Returns the number of CA distinguished names received from the peer in the TLS 1.3 certificate_authorities extension. This is typically called inside a cert_cb (WOLFSSL_CERT_SETUP_CB) on the server side to inspect which CAs the client trusts.

Parameters:

- **ssl** pointer to a WOLFSSL object, created with wolfSSL_new().

See:

- wolfSSL_GetPeerCertificateAuthority
- wolfSSL_UseCertificateAuthority

Return: >= 0 The number of peer CA DNs. Returns 0 if ssl is NULL or no certificate_authorities extension was received.

Requires WOLFSSL_TLS13 and !NO_CERTS and !WOLFSSL_NO_CA_NAMES.

Example

```
int count = wolfSSL_GetPeerCertificateAuthorityCount(ssl);
for (int i = 0; i < count; i++) {
    int sz = wolfSSL_GetPeerCertificateAuthority(ssl, i, NULL, 0);
    // sz is the DN size in bytes
}
```

```
int wolfSSL_GetPeerCertificateAuthority(
    const WOLFSSL * ssl,
    int idx,
    unsigned char * outDn,
    unsigned int outDnSz
)
```

Copies the idx-th CA distinguished name received from the peer in the TLS 1.3 certificate_authorities extension into the caller's buffer. The DN is the inner content of the DER-encoded Name (without the SEQUENCE header), matching the format accepted by wolfSSL_UseCertificateAuthority().

Parameters:

- **ssl** pointer to a WOLFSSL object, created with wolfSSL_new().
- **idx** zero-based index of the peer CA DN to retrieve. Must be less than the count returned by wolfSSL_GetPeerCertificateAuthorityCount().
- **outDn** output buffer to receive the DN bytes, or NULL to query size.
- **outDnSz** size of the output buffer in bytes.

See:

- wolfSSL_GetPeerCertificateAuthorityCount
- wolfSSL_UseCertificateAuthority
- wc_GetDecodedCertSubjectRaw

Return:

- 0 The number of bytes written to outDn, or the DN size if outDn is NULL.
- BAD_FUNC_ARG if ssl is NULL or idx is out of range.
- BUFFER_E if outDnSz is smaller than the DN.

If outDn is NULL, returns the size of the DN in bytes (allowing the caller to allocate the right amount of memory). If outDn is non-NULL and outDnSz is large enough, copies the DN bytes and returns the number of bytes written. If outDnSz is too small, returns BUFFER_E.

Requires WOLFSSL_TLS13 and !NO_CERTS and !WOLFSSL_NO_CA_NAMES.

Example

```
int count = wolfSSL_GetPeerCertificateAuthorityCount(ssl);
for (int i = 0; i < count; i++) {
    int sz = wolfSSL_GetPeerCertificateAuthority(ssl, i, NULL, 0);
    if (sz > 0) {
        unsigned char* dn = malloc(sz);
        wolfSSL_GetPeerCertificateAuthority(ssl, i, dn,
                (unsigned int)sz);
        // use dn[0..sz-1]
        free(dn);
    }
}
```

B wolfCrypt API Reference

B.1 ASN.1

B.1.1 Functions

	Name
int	**wc_BerToDer**(const byte * ber, word32 berSz, byte * der, word32 * derSz)This function converts BER (Basic Encoding Rules) formatted data to DER (Distinguished Encoding Rules) format. BER allows indefinite length encoding while DER requires definite lengths. This function calculates definite lengths for all indefinite length items.
void	**FreeAltNames**(DNS_entry * altNames, void * heap)This function frees a linked list of alternative names (DNS_entry structures). It deallocates each node and its associated name string, IP string, and RID string if present.
int	**wc_SetUnknownExtCallbackEx**(DecodedCert * cert, wc_UnknownExtCallbackEx cb, void * ctx)This function sets an extended callback for handling unknown certificate extensions during certificate parsing. The callback receives additional context information compared to the basic callback.
int	**wc_CheckCertSignature**(const byte * cert, word32 certSz, void * heap, WOLFSSL_CERT_MANAGER * cm)This function verifies the signature on a certificate using a certificate manager. It checks that the certificate is properly signed by a trusted CA.
int	**wc_EncodeObjectId**(const word16 * in, word32 inSz, byte * out, word32 * outSz)This function encodes an array of word16 values into an ASN.1 Object Identifier (OID) in DER format. OIDs are used to identify algorithms, extensions, and other objects in certificates and cryptographic protocols.
word32	**SetAlgoID**(int algoOID, byte * output, int type, int curveSz)This function sets the algorithm identifier in DER format. It encodes the algorithm OID and optional parameters based on the algorithm type and curve size.
int	**wc_DhPublicKeyDecode**(const byte * input, word32 * inOutIdx, DhKey * key, word32 inSz)This function decodes a DER encoded Diffie-Hellman public key. It extracts the public key value from the DER encoding and stores it in the DhKey structure.

	Name
int	**wc_InitCert**(Cert * cert)This function initializes a default cert, with the default options: version = 3 (0x2), serial = 0, sigType = SHA_WITH_RSA, issuer = blank, daysValid = 500, selfSigned = 1 (true) use subject as issuer, subject = blank.
Cert *	**wc_CertNew**(void * heap)This function allocates a new Cert structure for use during cert operations without the application having to allocate the structure itself. The Cert structure is also initialized by this function thus removing the need to call wc_InitCert(). When the application is finished using the allocated Cert structure wc_CertFree() must be called.
int	**wc_InitCert_ex**(Cert * cert, void * heap, int devId)Initializes certificate with heap hint and device ID.
void	**wc_CertFree**(Cert * cert)This function frees the memory allocated for a cert structure by a previous call to wc_CertNew().
int	**wc_MakeCert**(Cert * cert, byte * derBuffer, word32 derSz, RsaKey * rsaKey, ecc_key * eccKey, WC_RNG * rng)Used to make CA signed certs. Called after the subject information has been entered. This function makes an x509 Certificate v3 RSA or ECC from a cert input. It then writes this cert to derBuffer. It takes in either an rsaKey or an eccKey to generate the certificate. The certificate must be initialized with wc_InitCert before this method is called.
int	**wc_MakeCert_ex**(Cert * cert, byte * derBuffer, word32 derSz, int keyType, void * key, WC_RNG * rng)Makes certificate with generic key type support.
int	**wc_MakeCertReq_ex**(Cert * cert, byte * derBuffer, word32 derSz, int keyType, void * key)Makes certificate request with generic key type support.
int	**wc_SignCert_ex**(int requestSz, int sType, byte * buf, word32 buffSz, int keyType, void * key, WC_RNG * rng)Signs certificate with generic key type support.
int	**wc_MakeSigWithBitStr**(byte * sig, int sigSz, int sType, byte * buf, word32 bufSz, int keyType, void * key, WC_RNG * rng)Makes signature with bit string encoding. This function is used for dual algorithm certificate signing, where an alternative signature is created using a secondary key algorithm (e.g., a post_quantum algorithm alongside a traditional algorithm).
int	**wc_GetCertDates**(Cert * cert, struct tm * before, struct tm * after)Gets certificate validity dates.

	Name
int	**wc_GetDateInfo**(const byte * certDate, int certDateSz, const byte ** date, byte * format, int * length)Extracts date information from certificate date field. This function parses an ASN.1 encoded date (including tag and length) and returns a pointer to the raw date value bytes, the ASN.1 time type, and the length of the date value.
int	**wc_GetDateAsCalendarTime**(const byte * date, int length, byte format, struct tm * timearg)Converts certificate date to calendar time structure.
int	**wc_MakeCertReq**(Cert * cert, byte * derBuffer, word32 derSz, RsaKey * rsaKey, ecc_key * eccKey)This function makes a certificate signing request using the input certificate and writes the output to derBuffer. It takes in either an rsaKey or an eccKey to generate the certificate request. wc_SignCert() will need to be called after this function to sign the certificate request. Please see the wolfCrypt test application (./wolfcrypt/test/test.c) for an example usage of this function.
int	**wc_SignCert**(int requestSz, int sigType, byte * derBuffer, word32 derSz, RsaKey * rsaKey, ecc_key * eccKey, WC_RNG * rng)This function signs buffer and adds the signature to the end of buffer. It takes in a signature type. Must be called after wc_MakeCert() or wc_MakeCertReq() if creating a CA signed cert.
int	**wc_MakeSelfCert**(Cert * cert, byte * derBuffer, word32 derSz, RsaKey * key, WC_RNG * rng)This function is a combination of the previous two functions, wc_MakeCert and wc_SignCert for self signing (the previous functions may be used for CA requests). It makes a certificate, and then signs it, generating a self-signed certificate.
int	**wc_SetIssuer**(Cert * cert, const char * issuerFile)This function sets the issuer for a certificate to the issuer in the provided pem issuerFile. It also changes the certificate's self-signed attribute to false. The issuer specified in issuerFile is verified prior to setting the cert issuer. This method is used to set fields prior to signing.
int	**wc_SetSubject**(Cert * cert, const char * subjectFile)This function sets the subject for a certificate to the subject in the provided pem subjectFile. This method is used to set fields prior to signing.

	Name
int	**wc_SetSubjectRaw**(Cert * cert, const byte * der, int derSz)This function sets the raw subject for a certificate from the subject in the provided der buffer. This method is used to set the raw subject field prior to signing.
int	**wc_GetSubjectRaw**(byte ** subjectRaw, Cert * cert)This function gets the raw subject from the certificate structure.
int	**wc_SetAltNames**(Cert * cert, const char * file)This function sets the alternate names for a certificate to the alternate names in the provided pem file. This is useful in the case that one wishes to secure multiple domains with the same certificate. This method is used to set fields prior to signing.
int	**wc_SetIssuerBuffer**(Cert * cert, const byte * der, int derSz)This function sets the issuer for a certificate from the issuer in the provided der buffer. It also changes the certificate's self-signed attribute to false. This method is used to set fields prior to signing.
int	**wc_SetIssuerRaw**(Cert * cert, const byte * der, int derSz)This function sets the raw issuer for a certificate from the issuer in the provided der buffer. This method is used to set the raw issuer field prior to signing.
int	**wc_SetSubjectBuffer**(Cert * cert, const byte * der, int derSz)This function sets the subject for a certificate from the subject in the provided der buffer. This method is used to set fields prior to signing.
int	**wc_SetAltNamesBuffer**(Cert * cert, const byte * der, int derSz)This function sets the alternate names for a certificate from the alternate names in the provided der buffer. This is useful in the case that one wishes to secure multiple domains with the same certificate. This method is used to set fields prior to signing.
int	**wc_SetDNSEntry**(void * heap, const char * str, int strLen, int type, DNS_entry ** entries)This function allocates a single subject alternative name (SAN) entry, copies the supplied name into it, sets its GeneralName type and length, and appends it to the linked list pointed to by entries. The name is duplicated internally, so the caller's buffer need not outlive the call. The resulting list can be encoded with wc_FlattenAltNames and must be freed with FreeAltNames.

	Name
int	**wc_FlattenAltNames**(byte * output, word32 outputSz, const DNS_entry * names)This function encodes a linked list of subject alternative name entries into the DER GeneralNames SEQUENCE used as the value of the subjectAltName certificate extension. The output is suitable for assigning to Cert.altNames (with the return value stored in Cert.altNamesSz) prior to signing.
int	**wc_SetAltNamesFromList**(Cert * cert, const DNS_entry * names)This function encodes a linked list of subject alternative name entries directly into a Cert structure, ready for signing. It is a convenience wrapper around wc_FlattenAltNames: the list is encoded into cert->altNames and the encoded length is stored in cert->altNamesSz, so the caller does not have to manage the buffer or size bookkeeping. The supplied list is not consumed and must still be freed by the caller with FreeAltNames.
int	**wc_SetDatesBuffer**(Cert * cert, const byte * der, int derSz)This function sets the dates for a certificate from the date range in the provided der buffer. This method is used to set fields prior to signing.
int	**wc_SetAuthKeyIdFromPublicKey**(Cert * cert, RsaKey * rsakey, ecc_key * eckey)Set AKID from either an RSA or ECC public key. note: Only set one of rsakey or eckey, not both.
int	**wc_SetAuthKeyIdFromPublicKey_ex**(Cert * cert, int keyType, void * key)Sets authority key ID from public key with generic key type.
int	**wc_SetAuthKeyIdFromCert**(Cert * cert, const byte * der, int derSz)Set AKID from from DER encoded certificate.
int	**wc_SetAuthKeyId**(Cert * cert, const char * file)Set AKID from certificate file in PEM format.
int	**wc_SetSubjectKeyIdFromPublicKey**(Cert * cert, RsaKey * rsakey, ecc_key * eckey)Set SKID from RSA or ECC public key.
int	**wc_SetSubjectKeyIdFromPublicKey_ex**(Cert * cert, int keyType, void * key)Sets subject key ID from public key with generic key type.
int	**wc_SetSubjectKeyId**(Cert * cert, const char * file)Set SKID from public key file in PEM format. Both arguments are required.
int	**wc_SetExtKeyUsage**(Cert * cert, const char * value)Sets extended key usage using comma-delimited string.

	Name
int	**wc_SetExtKeyUsageOID**(Cert * cert, const char * oid, word32 sz, byte idx, void * heap)Sets extended key usage using OID string.
int	**wc_PemPubKeyToDer**(const char * fileName, unsigned char * derBuf, int derSz)Loads a PEM key from a file and converts to a DER encoded buffer.
int	**wc_PemPubKeyToDer_ex**(const char * fileName, DerBuffer ** der)Loads PEM public key from file to DER buffer.
int	**wc_PubKeyPemToDer**(const unsigned char * pem, int pemSz, unsigned char * buff, int buffSz)Convert a PEM encoded public key to DER. Returns the number of bytes written to the buffer or a negative value for an error.
int	**wc_PemGetHeaderFooter**(int type, const char ** header, const char ** footer)Gets PEM header and footer strings for given type.
int	**wc_AllocDer**(DerBuffer ** pDer, word32 length, int type, void * heap)Allocates DER buffer with specified length and type.
void	**wc_FreeDer**(DerBuffer ** pDer)Frees DER buffer allocated by wc_AllocDer or wc_PemToDer.
int	**wc_PemToDer**(const unsigned char * buff, long longSz, int type, DerBuffer ** pDer, void * heap, EncryptedInfo * info, int * keyFormat)Converts PEM to DER format with encryption info support.
int	**wc_PemCertToDer**(const char * fileName, unsigned char * derBuf, int derSz)This function converts a pem certificate to a der certificate, and places the resulting certificate in the derBuf buffer provided.
int	**wc_DerToPem**(const byte * der, word32 derSz, byte * output, word32 outSz, int type)This function converts a der formatted input certificate, contained in the der buffer, into a pem formatted output certificate, contained in the output buffer. It should be noted that this is not an in place conversion, and a separate buffer must be utilized to store the pem formatted output.

	Name
int	**wc_DerToPemEx**(const byte * der, word32 derSz, byte * output, word32 outSz, byte * cipher_info, int type)This function converts a der formatted input certificate, contained in the der buffer, into a pem formatted output certificate, contained in the output buffer. It should be noted that this is not an in place conversion, and a separate buffer must be utilized to store the pem formatted output. Allows setting cipher info.
int	**wc_PemCertToDer_ex**(const char * fileName, DerBuffer ** der)Loads PEM certificate from file to DER buffer.
word32	**wc_PkcsPad**(byte * buf, word32 sz, word32 blockSz)Adds PKCS padding to buffer for RSA encryption.
int	**wc_EccPrivateKeyDecode**(const byte * input, word32 * inOutIdx, ecc_key * key, word32 inSz)This function reads in an ECC private key from the input buffer, input, parses the private key, and uses it to generate an ecc_key object, which it stores in key.
int	**wc_EccKeyToDer**(ecc_key * key, byte * output, word32 outLen)This function writes a private ECC key to der format.
int	**wc_EccPublicKeyDecode**(const byte * input, word32 * inOutIdx, ecc_key * key, word32 inSz)Decodes an ECC public key from an input buffer. It will parse an ASN sequence to retrieve the ECC key.
int	**wc_EccPublicKeyToDer**(ecc_key * key, byte * output, word32 outLen, int with_AlgCurve)This function converts the ECC public key to DER format. It returns the size of buffer used. The public ECC key in DER format is stored in output buffer. The with_AlgCurve flag will include a header that has the Algorithm and Curve information.
int	**wc_EccPublicKeyToDer_ex**(ecc_key * key, byte * output, word32 outLen, int with_AlgCurve, int comp)This function converts the ECC public key to DER format. It returns the size of buffer used. The public ECC key in DER format is stored in output buffer. The with_AlgCurve flag will include a header that has the Algorithm and Curve information. The comp parameter determines if the public key will be exported as compressed.

	Name
int	**wc_Curve25519PrivateKeyDecode**(const byte * input, word32 * inOutIdx, curve25519_key * key, word32 inSz)This function decodes a Curve25519 private key (only) from a DER encoded buffer.
int	**wc_Curve25519PublicKeyDecode**(const byte * input, word32 * inOutIdx, curve25519_key * key, word32 inSz)This function decodes a Curve25519 public key (only) from a DER encoded buffer.
int	**wc_Curve25519KeyDecode**(const byte * input, word32 * inOutIdx, curve25519_key * key, word32 inSz)This function decodes a Curve25519 key from a DER encoded buffer. It can decode either a private key, a public key, or both.
int	**wc_Curve25519PrivateKeyToDer**(curve25519_key * key, byte * output, word32 outLen)This function encodes a Curve25519 private key to DER format. If the input key structure contains a public key, it will be ignored.
int	**wc_Curve25519PublicKeyToDer**(curve25519_key * key, byte * output, word32 outLen, int withAlg)This function encodes a Curve25519 public key to DER format. If the input key structure contains a private key, it will be ignored.
int	**wc_Curve25519KeyToDer**(curve25519_key * key, byte * output, word32 outLen, int withAlg)This function encodes a Curve25519 key to DER format. It can encode either a private key, a public key, or both.
word32	**wc_EncodeSignature**(byte * out, const byte * digest, word32 digSz, int hashOID)This function encodes a digital signature into the output buffer, and returns the size of the encoded signature created.
int	**wc_GetCTC_HashOID**(int type)This function returns the hash OID that corresponds to a hashing type. For example, when given the type: WC_SHA512, this function returns the identifier corresponding to a SHA512 hash, SHA512h.
void	**wc_SetCert_Free**(Cert * cert)This function cleans up memory and resources used by the certificate structure's decoded cert cache. When WOLFSSL_CERT_GEN_CACHE is defined the decoded cert structure is cached in the certificate structure. This allows subsequent calls to certificate set functions to avoid parsing the decoded cert on each call.

COPYRIGHT ©2024 wolfSSL Inc.

	Name
int	**wc_GetPkcs8TraditionalOffset**(byte * input, word32 * inOutIdx, word32 sz)This function finds the beginning of the traditional private key inside a PKCS#8 unencrypted buffer.
int	**wc_CreatePKCS8Key**(byte * out, word32 * outSz, byte * key, word32 keySz, int algoID, const byte * curveOID, word32 oidSz)This function takes in a DER private key and converts it to PKCS#8 format. Also used in creating PKCS#12 shrouded key bags. See RFC 5208.
int	**wc_EncryptPKCS8Key**(byte * key, word32 keySz, byte * out, word32 * outSz, const char * password, int passwordSz, int vPKCS, int pbeOid, int encAlgId, byte * salt, word32 saltSz, int itt, WC_RNG * rng, void * heap)This function takes in an unencrypted PKCS#8 DER key (e.g. one created by wc_CreatePKCS8Key) and converts it to PKCS#8 encrypted format. The resulting encrypted key can be decrypted using wc_DecryptPKCS8Key. See RFC 5208.
int	**wc_EncryptPKCS8Key_ex**(byte * key, word32 keySz, byte * out, word32 * outSz, const char * password, int passwordSz, int vPKCS, int pbeOid, int encAlgId, byte * salt, word32 saltSz, int itt, WC_RNG * rng, void * heap, int devId)Encrypts PKCS#8 key with extended parameters.
int	**wc_GetTime**(void * timePtr, word32 timeSize)Gets current time for certificate operations.
int	**wc_EncryptedInfoGet**(EncryptedInfo * info, const char * cipherName)Gets encryption info from encrypted PEM.
int	**wc_ParseCertPIV**(wc_CertPIV * cert, const byte * buf, word32 totalSz)Parses PIV certificate format.
int	**wc_GetSubjectPubKeyInfoDerFromCert**(const byte * certDer, word32 certDerSz, byte * pubKeyDer, word32 * pubKeyDerSz)Extracts subject public key info from certificate.
int	**wc_GetDecodedCertSubject**(const struct DecodedCert * cert, char * buf, word32 * bufSz)Retrieves the subject name from a decoded certificate.
int	**wc_GetDecodedCertIssuer**(const struct DecodedCert * cert, char * buf, word32 * bufSz)Retrieves the issuer name from a decoded certificate.

	Name
int	**wc_GetDecodedCertSerial**(const struct DecodedCert * cert, byte * buf, word32 * bufSz)Retrieves the serial number from a decoded certificate.
int	**wc_GetUUIDFromCert**(struct DecodedCert * cert, byte * uuid, int * uuidSz)Extracts UUID from certificate.
int	**wc_GetFASCNFromCert**(struct DecodedCert * cert, byte * fascn, int * fascnSz)Extracts FASCN from certificate.
int	**wc_GeneratePreTBS**(struct DecodedCert * cert, byte * der, int derSz)Generates the pre_TBS (To Be Signed) certificate data from a decoded certificate. The TBS portion is the certificate data that gets signed by the certificate authority. This function is used in dual algorithm certificate creation where the TBS data needs to be extracted for signing with an alternative algorithm (e.g., a post_quantum algorithm).
void	**wc_InitDecodedAcert**(struct DecodedAcert * acert, void * heap)Initializes decoded attribute certificate structure.
void	**wc_FreeDecodedAcert**(struct DecodedAcert * acert)Frees decoded attribute certificate structure.
int	**wc_ParseX509Acert**(struct DecodedAcert * acert, int verify)Parses X.509 attribute certificate.
int	**wc_VerifyX509Acert**(const byte * acert, word32 acertSz, const byte * issuerCert, word32 issuerCertSz, void * cm)Verifies X.509 attribute certificate.
int	**wc_DecryptPKCS8Key**(byte * input, word32 sz, const char * password, int passwordSz)This function takes an encrypted PKCS#8 DER key and decrypts it to PKCS#8 unencrypted DER.Undoes the encryption done by wc_EncryptPKCS8Key. See RFC5208. The input buffer is overwritten with the decrypted data.
int	**wc_CreateEncryptedPKCS8Key**(byte * key, word32 keySz, byte * out, word32 * outSz, const char * password, int passwordSz, int vPKCS, int pbeOid, int encAlgId, byte * salt, word32 saltSz, int itt, WC_RNG * rng, void * heap)This function takes a traditional, DER key, converts it to PKCS#8 format, and encrypts it. It uses wc_CreatePKCS8Key and wc_EncryptPKCS8Key to do this.

	Name
void	**wc_InitDecodedCert**(struct DecodedCert * cert, const byte * source, word32 inSz, void * heap)This function initializes the DecodedCert pointed to by the “cert” parameter. It saves the “source” pointer to a DER-encoded certificate of length “inSz.” This certificate can be parsed by a subsequent call to wc_ParseCert.
int	**wc_ParseCert**(DecodedCert * cert, int type, int verify, WOLFSSL_CERT_MANAGER * cm)This function parses the DER-encoded certificate saved in the DecodedCert object and populates the fields of that object. The DecodedCert must have been initialized with a prior call to wc_InitDecodedCert. This function takes an optional pointer to a CertificateManager object, which is used to populate the certificate authority information of the DecodedCert, if the CA is found in the CertificateManager.
void	**wc_FreeDecodedCert**(struct DecodedCert * cert)This function frees a DecodedCert that was previously initialized with wc_InitDecodedCert.
int	**wc_SetTimeCb**(wc_time_cb f)This function registers a time callback that will be used anytime wolfSSL needs to get the current time. The prototype of the callback should be the same as the “time” function from the C standard library.
time_t	**wc_Time**(time_t * t)This function gets the current time. By default, it uses the XTIME macro, which varies between platforms. The user can use a function of their choosing instead via the wc_SetTimeCb function.
int	**wc_SetCustomExtension**(Cert * cert, int critical, const char * oid, const byte * der, word32 derSz)This function injects a custom extension in to an X.509 certificate. note: The content at the address pointed to by any of the parameters that are pointers must not be modified until the certificate is generated and you have the der output. This function does NOT copy the contents to another buffer.
int	**wc_SetUnknownExtCallback**(DecodedCert * cert, wc_UnknownExtCallback cb)This function registers a callback that will be used anytime wolfSSL encounters an unknown X.509 extension in a certificate while parsing a certificate. The prototype of the callback should be:

	Name
int	**wc_CheckCertSigPubKey**(const byte * cert, word32 certSz, void * heap, const byte * pubKey, word32 pubKeySz, int pubKeyOID)This function verifies the signature in the der form of an X.509 certificate against a public key. The public key is expected to be the full subject public key info in der form.
int	**wc_Asn1PrintOptions_Init**(Asn1PrintOptions * opts)This function initializes the ASN.1 print options.
int	**wc_Asn1PrintOptions_Set**(Asn1PrintOptions * opts, enum Asn1PrintOpt opt, word32 val)This function sets a print option into an ASN.1 print options object.
int	**wc_Asn1_Init**(Asn1 * asn1)This function initializes an ASN.1 parsing object.
int	**wc_Asn1_SetFile**(Asn1 * asn1, XFILE file)This function sets the file to use when printing into an ASN.1 parsing object.
int	**wc_Asn1_PrintAll**(Asn1 * asn1, Asn1PrintOptions * opts, unsigned char * data, word32 len)Print all ASN.1 items.
int	**wc_Asn1_SetOidToNameCb**(Asn1 * asn1, Asn1OidToNameCb nameCb)Sets OID to name callback for ASN.1 parsing.
int	**wc_GetDecodedCertSubjectRaw**(const struct DecodedCert * cert, const byte ** subjectRaw, int * subjectRawSz)Retrieves the raw DER-encoded subject Name content from a parsed DecodedCert.

B.1.2 Functions Documentation

```
int wc_BerToDer(
    const byte * ber,
    word32 berSz,
    byte * der,
    word32 * derSz
)
```

This function converts BER (Basic Encoding Rules) formatted data to DER (Distinguished Encoding Rules) format. BER allows indefinite length encoding while DER requires definite lengths. This function calculates definite lengths for all indefinite length items.

Parameters:

- **ber** pointer to the buffer containing BER formatted data
- **berSz** size of the BER data in bytes
- **der** pointer to buffer to store DER formatted data (can be NULL to calculate required size)
- **derSz** pointer to size of der buffer; updated with actual size needed or used

See: wc_EncodeObjectId

Return:

- 0 On success.
- ASN_PARSE_E If the BER data is invalid.
- BAD_FUNC_ARG If ber or derSz are NULL.
- BUFFER_E If der is not NULL and derSz is too small.

Note: This API is not public by default. Define WOLFSSL_PUBLIC_ASN to expose APIs marked WOLFSSL_ASN_API.

Example

```
byte ber[256] = { }; // BER encoded data
byte der[256];
word32 derSz = sizeof(der);

int ret = wc_BerToDer(ber, sizeof(ber), der, &derSz);
if (ret == 0) {
    // der now contains DER formatted data of length derSz
}
```

```
void FreeAltNames(
    DNS_entry * altNames,
    void * heap
)
```

This function frees a linked list of alternative names (DNS_entry structures). It deallocates each node and its associated name string, IP string, and RID string if present.

Parameters:

- **altNames** pointer to the head of the alternative names linked list
- **heap** pointer to heap hint for memory deallocation (can be NULL)

See: AltNameNew

Return: none No return value.

Note: This API is not public by default. Define WOLFSSL_PUBLIC_ASN to expose APIs marked WOLFSSL_ASN_API.

Example

```
DNS_entry* altNames = NULL;
// populate altNames with certificate alternative names

FreeAltNames(altNames, NULL);
// altNames list is now freed
```

```
int wc_SetUnknownExtCallbackEx(
    DecodedCert * cert,
    wc_UnknownExtCallbackEx cb,
    void * ctx
)
```

This function sets an extended callback for handling unknown certificate extensions during certificate parsing. The callback receives additional context information compared to the basic callback.

Parameters:

- **cert** pointer to the DecodedCert structure
- **cb** callback function to handle unknown extensions
- **ctx** context pointer passed to the callback

See:

- wc_SetUnknownExtCallback
- wc_InitDecodedCert

Return:

- 0 On success.
- BAD_FUNC_ARG If cert is NULL.

Note: This API is not public by default. Define WOLFSSL_PUBLIC_ASN to expose APIs marked WOLFSSL_ASN_API.

Example

```
DecodedCert cert;

int UnknownExtCallback(const byte* oid, word32 oidSz, int crit,
                       const byte* der, word32 derSz, void* ctx) {
    // handle unknown extension
    return 0;
}

wc_InitDecodedCert(&cert, derCert, derCertSz, NULL);
wc_SetUnknownExtCallbackEx(&cert, UnknownExtCallback, myContext);
wc_ParseCert(&cert, CERT_TYPE, NO_VERIFY, NULL);
```

```
int wc_CheckCertSignature(
    const byte * cert,
    word32 certSz,
    void * heap,
    WOLFSSL_CERT_MANAGER * cm
)
```

This function verifies the signature on a certificate using a certificate manager. It checks that the certificate is properly signed by a trusted CA.

Parameters:

- **cert** pointer to the DER encoded certificate
- **certSz** size of the certificate in bytes
- **heap** pointer to heap hint for memory allocation (can be NULL)
- **cm** pointer to certificate manager containing trusted CAs

See:

- wolfSSL_CertManagerNew
- wolfSSL_CertManagerLoadCA

Return:

- 0 On successful signature verification.
- ASN_SIG_CONFIRM_E If signature verification fails.
- Other negative values on error.

Example

```
byte cert[2048] = { }; // DER encoded certificate
word32 certSz = sizeof(cert);
WOLFSSL_CERT_MANAGER* cm;

cm = wolfSSL_CertManagerNew();
wolfSSL_CertManagerLoadCA(cm, "ca-cert.pem", NULL);

int ret = wc_CheckCertSignature(cert, certSz, NULL, cm);
if (ret == 0) {
    // certificate signature is valid
}
wolfSSL_CertManagerFree(cm);
```

```
int wc_EncodeObjectId(
    const word16 * in,
    word32 inSz,
    byte * out,
    word32 * outSz
)
```

This function encodes an array of word16 values into an ASN.1 Object Identifier (OID) in DER format. OIDs are used to identify algorithms, extensions, and other objects in certificates and cryptographic protocols.

Parameters:

- **in** pointer to array of word16 values representing OID components
- **inSz** number of components in the OID
- **out** pointer to buffer to store encoded OID (can be NULL to calculate size)

- **outSz** pointer to size of out buffer; updated with actual size

See: wc_BerToDer

Return:

- 0 On success.
- BAD_FUNC_ARG If in, inSz, or outSz are invalid.
- BUFFER_E If out is not NULL and outSz is too small.

Example

```
word16 oid[] = {1, 2, 840, 113549, 1, 1, 11}; // sha256WithRSAEncryption
byte encoded[32];
word32 encodedSz = sizeof(encoded);

int ret = wc_EncodeObjectId(oid, sizeof(oid)/sizeof(word16),
                            encoded, &encodedSz);
if (ret == 0) {
    // encoded contains DER encoded OID
}
```

```
word32 SetAlgoID(
    int algoOID,
    byte * output,
    int type,
    int curveSz
)
```

This function sets the algorithm identifier in DER format. It encodes the algorithm OID and optional parameters based on the algorithm type and curve size.

Parameters:

- **algoOID** algorithm object identifier constant
- **output** pointer to buffer to store encoded algorithm ID
- **type** type of encoding (oidSigType, oidHashType, etc.)
- **curveSz** size of the curve for ECC algorithms (0 for non-ECC)

See: wc_EncodeObjectId

Return:

- Length of the encoded algorithm identifier on success.
- Negative value on error.

Example

```
byte algId[32];
word32 len;
```

```
len = SetAlgoID(CTC_SHA256wRSA, algId, oidSigType, 0);
if (len > 0) {
    // algId contains encoded algorithm identifier
}
```

```
int wc_DhPublicKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    DhKey * key,
    word32 inSz
)
```

This function decodes a DER encoded Diffie-Hellman public key. It extracts the public key value from the DER encoding and stores it in the DhKey structure.

Parameters:

- **input** pointer to buffer containing DER encoded public key
- **inOutIdx** pointer to index in buffer; updated to end of key
- **key** pointer to DhKey structure to store decoded public key
- **inSz** size of the input buffer

See:

- wc_InitDhKey
- wc_DhKeyDecode

Return:

- 0 On success.
- BAD_FUNC_ARG If input, inOutIdx, key, or inSz are invalid.
- ASN_PARSE_E If the DER encoding is invalid.
- Other negative values on error.

Example

```
byte derKey[256] = { }; // DER encoded DH public key
word32 idx = 0;
DhKey key;

wc_InitDhKey(&key);
int ret = wc_DhPublicKeyDecode(derKey, &idx, &key, sizeof(derKey));
if (ret == 0) {
    // key now contains the decoded public key
}
wc_FreeDhKey(&key);
```

```
int wc_InitCert(
    Cert * cert
)
```

This function initializes a default cert, with the default options: version = 3 (0x2), serial = 0, sigType = SHA_WITH_RSA, issuer = blank, daysValid = 500, selfSigned = 1 (true) use subject as issuer, subject = blank.

Parameters:

- **cert** pointer to an uninitialized cert structure to initialize

See:

- wc_MakeCert
- wc_MakeCertReq

Return: none No returns.

Example

```
Cert myCert;
wc_InitCert(&myCert);
```

```
Cert * wc_CertNew(
    void * heap
)
```

This function allocates a new Cert structure for use during cert operations without the application having to allocate the structure itself. The Cert structure is also initialized by this function thus removing the need to call wc_InitCert(). When the application is finished using the allocated Cert structure wc_CertFree() must be called.

Parameters:

- **A** pointer to the heap used for dynamic allocation. Can be NULL.

See:

- wc_InitCert
- wc_MakeCert
- wc_CertFree

Return:

- pointer If successful the call will return a pointer to the newly allocated and initialized Cert.
- NULL On a memory allocation failure.

Example

```
Cert*   myCert;

myCert = wc_CertNew(NULL);
if (myCert == NULL) {
    // Cert creation failure
}
```

```
int wc_InitCert_ex(
    Cert * cert,
    void * heap,
    int devId
)
```

Initializes certificate with heap hint and device ID.

Parameters:

- **cert** Cert structure to initialize
- **heap** Heap hint for memory allocation
- **devId** Device ID for hardware acceleration

See:

- wc_InitCert
- wc_MakeCert_ex

Return:

- 0 on success
- BAD_FUNC_ARG if cert is NULL

Example

```
Cert myCert;
int ret = wc_InitCert_ex(&myCert, NULL, INVALID_DEVID);
if (ret != 0) {
    // error initializing cert
}
```

```
void wc_CertFree(
    Cert * cert
)
```

This function frees the memory allocated for a cert structure by a previous call to wc_CertNew().

Parameters:

- **A** pointer to the cert structure to free.

See:

- wc_InitCert
- wc_MakeCert
- wc_CertNew

Return: None.

Example

```
Cert*   myCert;

myCert = wc_CertNew(NULL);

// Perform cert operations.

wc_CertFree(myCert);
```

```
int wc_MakeCert(
    Cert * cert,
    byte * derBuffer,
    word32 derSz,
    RsaKey * rsaKey,
    ecc_key * eccKey,
    WC_RNG * rng
)
```

Used to make CA signed certs. Called after the subject information has been entered. This function makes an x509 Certificate v3 RSA or ECC from a cert input. It then writes this cert to derBuffer. It takes in either an rsaKey or an eccKey to generate the certificate. The certificate must be initialized with wc_InitCert before this method is called.

Parameters:

- **cert** pointer to an initialized cert structure
- **derBuffer** pointer to the buffer in which to hold the generated cert
- **derSz** size of the buffer in which to store the cert
- **rsaKey** pointer to an RsaKey structure containing the rsa key used to generate the certificate
- **eccKey** pointer to an EccKey structure containing the ecc key used to generate the certificate
- **rng** pointer to the random number generator used to make the cert

See:

- wc_InitCert
- wc_MakeCertReq

Return:

- Success On successfully making an x509 certificate from the specified input cert, returns the size of the cert generated.

- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- BUFFER_E Returned if the provided derBuffer is too small to store the generated certificate
- BAD_FUNC_ARG Returned if cert->serialSz is negative, if the encoded serial would exceed CTC_SERIAL_SIZE octets, or if the serial number is zero. RFC 5280 4.1.2.2 requires a positive serial of at most 20 octets; define WOLFSSL_ASN_ALLOW_0_SERIAL to permit a zero serial.
- Others Additional error messages may be returned if the cert generation is not successful.

A serial number left at the wc_InitCert default (cert->serialSz of 0) is randomly generated. A caller-supplied serial is taken as a big-endian value of at most CTC_SERIAL_SIZE bytes and is normalized to a minimal DER INTEGER: redundant leading zero bytes are stripped and the sign pad is added back when the high bit is set. Supplying the magnitude alone is enough; a sign pad the caller adds is accepted and is not duplicated. The encoded value, including any sign pad, must not exceed CTC_SERIAL_SIZE octets.

Example

```
Cert myCert;
wc_InitCert(&myCert);
WC_RNG rng;
//initialize rng;
RsaKey key;
//initialize key;
byte * derCert = malloc(FOURK_BUF);
word32 certSz;
certSz = wc_MakeCert(&myCert, derCert, FOURK_BUF, &key, NULL, &rng);
```

```
int wc_MakeCert_ex(
    Cert * cert,
    byte * derBuffer,
    word32 derSz,
    int keyType,
    void * key,
    WC_RNG * rng
)
```

Makes certificate with generic key type support.

Parameters:

- **cert** Initialized cert structure
- **derBuffer** Buffer for generated certificate
- **derSz** Size of derBuffer
- **keyType** Key type (RSA_TYPE, ECC_TYPE, ED25519_TYPE, etc.)
- **key** Pointer to key structure
- **rng** Random number generator

See:

- wc_MakeCert
- wc_SignCert_ex

Return:

- Size of certificate on success
- MEMORY_E if memory allocation fails
- BUFFER_E if buffer too small
- BAD_FUNC_ARG if the serial number is zero, negative in size, or longer than CTC_SERIAL_SIZE
- Other error codes on failure

The serial number contract is the same as wc_MakeCert().

Example

```
Cert myCert;
wc_InitCert(&myCert);
byte derCert[4096];
RsaKey key;
WC_RNG rng;
int certSz = wc_MakeCert_ex(&myCert, derCert, sizeof(derCert),
                            RSA_TYPE, &key, &rng);
```

```
int wc_MakeCertReq_ex(
    Cert * cert,
    byte * derBuffer,
    word32 derSz,
    int keyType,
    void * key
)
```

Makes certificate request with generic key type support.

Parameters:

- **cert** Initialized cert structure
- **derBuffer** Buffer for generated certificate request
- **derSz** Size of derBuffer
- **keyType** Key type (RSA_TYPE, ECC_TYPE, ED25519_TYPE, etc.)
- **key** Pointer to key structure

See:

- wc_MakeCertReq
- wc_SignCert_ex

Return:

- Size of certificate request on success
- MEMORY_E if memory allocation fails
- BUFFER_E if buffer too small
- Other error codes on failure

Example

```
Cert myCert;
wc_InitCert(&myCert);
byte derCert[4096];
EccKey key;
int certSz = wc_MakeCertReq_ex(&myCert, derCert, sizeof(derCert),
                               ECC_TYPE, &key);
```

```
int wc_SignCert_ex(
    int requestSz,
    int sType,
    byte * buf,
    word32 buffSz,
    int keyType,
    void * key,
    WC_RNG * rng
)
```

Signs certificate with generic key type support.

Parameters:

- **requestSz** Size of certificate body to sign
- **sType** Signature type
- **buf** Buffer containing certificate to sign
- **buffSz** Total size of buffer
- **keyType** Key type (RSA_TYPE, ECC_TYPE, ED25519_TYPE, etc.)
- **key** Pointer to key structure
- **rng** Random number generator

See:

- wc_SignCert
- wc_MakeCert_ex

Return:

- New size of certificate with signature on success
- MEMORY_E if memory allocation fails
- BUFFER_E if buffer too small
- Other error codes on failure

Example

```
Cert myCert;
byte derCert[4096];
RsaKey key;
WC_RNG rng;
```

```
// Initialize cert and set fields (issuer, subject, dates, etc.)
wc_InitCert(&myCert);
// ... set myCert fields ...
// Generate certificate body (TBS - To Be Signed)
int bodySz = wc_MakeCert_ex(&myCert, derCert, sizeof(derCert),
                            RSA_TYPE, &key, &rng);
if (bodySz > 0) {
    // bodySz is the size of the unsigned certificate body
    // Sign the certificate body and append signature
    int certSz = wc_SignCert_ex(bodySz, CTC_SHA256wRSA,
                                derCert, sizeof(derCert), RSA_TYPE,
                                &key, &rng);
    // derCert now contains complete signed certificate of size certSz
}
```

```
int wc_MakeSigWithBitStr(
    byte * sig,
    int sigSz,
    int sType,
    byte * buf,
    word32 bufSz,
    int keyType,
    void * key,
    WC_RNG * rng
)
```

Makes signature with bit string encoding. This function is used for dual algorithm certificate signing, where an alternative signature is created using a secondary key algorithm (e.g., a post-quantum algorithm alongside a traditional algorithm).

Parameters:

- **sig** Output buffer for signature
- **sigSz** Size of signature buffer
- **sType** Signature type
- **buf** Data to sign (typically the TBS - To Be Signed - certificate data)
- **bufSz** Size of data
- **keyType** Key type (RSA_TYPE, ECC_TYPE, ED25519_TYPE, etc.)
- **key** Pointer to key structure
- **rng** Random number generator

See:

- wc_SignCert_ex
- wc_GeneratePreTBS

Return:

- Size of signature on success
- Negative on error

Note: This API is only available when WOLFSSL_DUAL_ALG_CERTS is defined, which enables support for dual algorithm certificates used in Post-Quantum cryptography to provide hybrid signing with both traditional and PQ algorithms.

Example

```
byte sig[512], data[256];
RsaKey key;
WC_RNG rng;
int sigSz = wc_MakeSigWithBitStr(sig, sizeof(sig), CTC_SHA256wRSA,
                                 data, sizeof(data), RSA_TYPE,
                                 &key, &rng);
```

```
int wc_GetCertDates(
    Cert * cert,
    struct tm * before,
    struct tm * after
)
```

Gets certificate validity dates.

Parameters:

- **cert** Certificate structure
- **before** Output for notBefore date
- **after** Output for notAfter date

See: wc_InitCert

Return:

- 0 on success
- BAD_FUNC_ARG if parameters invalid

Example

```
Cert myCert;
struct tm beforeDate, afterDate;
int ret = wc_GetCertDates(&myCert, &beforeDate, &afterDate);
```

```
int wc_GetDateInfo(
    const byte * certDate,
    int certDateSz,
    const byte ** date,
    byte * format,
    int * length
)
```

Extracts date information from certificate date field. This function parses an ASN.1 encoded date (including tag and length) and returns a pointer to the raw date value bytes, the ASN.1 time type, and the length of the date value.

Parameters:

- **certDate** Certificate date buffer containing ASN.1 encoded date (tag + length + value)
- **certDateSz** Size of certificate date buffer
- **date** Output pointer set to the raw date value bytes (without tag/length)
- **format** Output byte indicating ASN.1 time type: ASN_UTC_TIME (0x17) or ASN_GENERALIZED_TIME (0x18)
- **length** Output length of the raw date value in bytes

See:

- wc_GetCertDates
- wc_GetDateAsCalendarTime

Return:

- 0 on success
- BAD_FUNC_ARG if parameters invalid
- ASN_PARSE_E if date parsing fails

Example

```
const byte* certDate;
const byte* date;
byte format;
int length;
int ret = wc_GetDateInfo(certDate, certDateSz, &date,
                         &format, &length);
if (ret == 0) {
    // date points to raw time bytes, format indicates UTC or
    // Generalized time, length is the number of date value bytes
}
```

```
int wc_GetDateAsCalendarTime(
    const byte * date,
    int length,
    byte format,
    struct tm * timearg
)
```

Converts certificate date to calendar time structure.

Parameters:

- **date** Date buffer
- **length** Length of date buffer

- **format** Date format (ASN_UTC_TIME or ASN_GENERALIZED_TIME)
- **timearg** Pointer to tm structure to fill

See:

- wc_GetDateInfo
- wc_GetCertDates

Return:

- 0 on success
- BAD_FUNC_ARG if parameters invalid
- ASN_TIME_E if time conversion fails

Example

```
const byte* date;
int length;
byte format;
struct tm timeInfo;
int ret = wc_GetDateAsCalendarTime(date, length, format,
                                   &timeInfo);
```

```
int wc_MakeCertReq(
    Cert * cert,
    byte * derBuffer,
    word32 derSz,
    RsaKey * rsaKey,
    ecc_key * eccKey
)
```

This function makes a certificate signing request using the input certificate and writes the output to derBuffer. It takes in either an rsaKey or an eccKey to generate the certificate request. wc_SignCert() will need to be called after this function to sign the certificate request. Please see the wolfCrypt test application (./wolfcrypt/test/test.c) for an example usage of this function.

Parameters:

- **cert** pointer to an initialized cert structure
- **derBuffer** pointer to the buffer in which to hold the generated certificate request
- **derSz** size of the buffer in which to store the certificate request
- **rsaKey** pointer to an RsaKey structure containing the rsa key used to generate the certificate request
- **eccKey** pointer to an EccKey structure containing the ecc key used to generate the certificate request

See:

- wc_InitCert
- wc_MakeCert

Return:

- Success On successfully making an X.509 certificate request from the specified input cert, returns the size of the certificate request generated.
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- BUFFER_E Returned if the provided derBuffer is too small to store the generated certificate
- Other Additional error messages may be returned if the certificate request generation is not successful.

Example

```
Cert myCert;
// initialize myCert
EccKey key;
//initialize key;
byte* derCert = (byte*)malloc(FOURK_BUF);

word32 certSz;
certSz = wc_MakeCertReq(&myCert, derCert, FOURK_BUF, NULL, &key);
```

```
int wc_SignCert(
    int requestSz,
    int sigType,
    byte * derBuffer,
    word32 derSz,
    RsaKey * rsaKey,
    ecc_key * eccKey,
    WC_RNG * rng
)
```

This function signs buffer and adds the signature to the end of buffer. It takes in a signature type. Must be called after wc_MakeCert() or wc_MakeCertReq() if creating a CA signed cert.

Parameters:

- **requestSz** the size of the certificate body we're requesting to have signed
- **sType** Type of signature to create. Valid options are: CTC_MD5wRSA, CTC_SHAwRSA, CTC_SHAwECDSA, CTC_SHA256wECDSA, and CTC_SHA256wRSA
- **buffer** pointer to the buffer containing the certificate to be signed. On success: will hold the newly signed certificate
- **buffSz** the (total) size of the buffer in which to store the newly signed certificate
- **rsaKey** pointer to an RsaKey structure containing the rsa key to used to sign the certificate
- **eccKey** pointer to an EccKey structure containing the ecc key to used to sign the certificate
- **rng** pointer to the random number generator used to sign the certificate

See:

- wc_InitCert
- wc_MakeCert

Return:

- Success On successfully signing the certificate, returns the new size of the cert (including signature).
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- BUFFER_E Returned if the provided buffer is too small to store the generated certificate
- Other Additional error messages may be returned if the cert generation is not successful.

Example

```
Cert myCert;
byte* derCert = (byte*)malloc(FOURK_BUF);
// initialize myCert, derCert
RsaKey key;
// initialize key;
WC_RNG rng;
// initialize rng

word32 certSz;
certSz = wc_SignCert(myCert.bodySz, myCert.sigType,derCert,FOURK_BUF,
&key, NULL,
&rng);
```

```
int wc_MakeSelfCert(
    Cert * cert,
    byte * derBuffer,
    word32 derSz,
    RsaKey * key,
    WC_RNG * rng
)
```

This function is a combination of the previous two functions, wc_MakeCert and wc_SignCert for self signing (the previous functions may be used for CA requests). It makes a certificate, and then signs it, generating a self-signed certificate.

Parameters:

- **cert** pointer to the cert to make and sign
- **buffer** pointer to the buffer in which to hold the signed certificate
- **buffSz** size of the buffer in which to store the signed certificate
- **key** pointer to an RsaKey structure containing the rsa key to used to sign the certificate
- **rng** pointer to the random number generator used to generate and sign the certificate

See:

- wc_InitCert
- wc_MakeCert
- wc_SignCert

Return:

- Success On successfully signing the certificate, returns the new size of the cert.
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- BUFFER_E Returned if the provided buffer is too small to store the generated certificate
- Other Additional error messages may be returned if the cert generation is not successful.

Example

```
Cert myCert;
byte* derCert = (byte*)malloc(FOURK_BUF);
// initialize myCert, derCert
RsaKey key;
// initialize key;
WC_RNG rng;
// initialize rng

word32 certSz;
certSz = wc_MakeSelfCert(&myCert, derCert, FOURK_BUF, &key, NULL, &rng);
```

```
int wc_SetIssuer(
    Cert * cert,
    const char * issuerFile
)
```

This function sets the issuer for a certificate to the issuer in the provided pem issuerFile. It also changes the certificate's self-signed attribute to false. The issuer specified in issuerFile is verified prior to setting the cert issuer. This method is used to set fields prior to signing.

Parameters:

- **cert** pointer to the cert for which to set the issuer
- **issuerFile** path of the file containing the pem formatted certificate

See:

- wc_InitCert
- wc_SetSubject
- wc_SetIssuerBuffer

Return:

- 0 Returned on successfully setting the issuer for the certificate
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_PARSE_E Returned if there is an error parsing the cert header file
- ASN_OBJECT_ID_E Returned if there is an error parsing the encryption type from the cert
- ASN_EXPECT_0_E Returned if there is a formatting error in the encryption specification of the cert file
- ASN_BEFORE_DATE_E Returned if the date is before the certificate start date

- ASN_AFTER_DATE_E Returned if the date is after the certificate expiration date
- ASN_BITSTR_E Returned if there is an error parsing a bit string from the certificate
- ECC_CURVE_OID_E Returned if there is an error parsing the ECC key from the certificate
- ASN_UNKNOWN_OID_E Returned if the certificate is using an unknown key object id
- ASN_VERSION_E Returned if the ALLOW_V1_EXTENSIONS option is not defined and the certificate is a V1 or V2 certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension
- ASN_CRIT_EXT_E Returned if an unfamiliar critical extension is encountered in processing the certificate
- ASN_SIG_OID_E Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
- ASN_SIG_CONFIRM_E Returned if confirming the certification signature fails
- ASN_NAME_INVALID_E Returned if the certificate's name is not permitted by the CA name constraints
- ASN_NO_SIGNER_E Returned if there is no CA signer to verify the certificate's authenticity

Example

```
Cert myCert;
// initialize myCert
if(wc_SetIssuer(&myCert, "./path/to/ca-cert.pem") != 0) {
    // error setting issuer
}
```

```
int wc_SetSubject(
    Cert * cert,
    const char * subjectFile
)
```

This function sets the subject for a certificate to the subject in the provided pem subjectFile. This method is used to set fields prior to signing.

Parameters:

- **cert** pointer to the cert for which to set the issuer
- **subjectFile** path of the file containing the pem formatted certificate

See:

- wc_InitCert
- wc_SetIssuer

Return:

- 0 Returned on successfully setting the issuer for the certificate
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_PARSE_E Returned if there is an error parsing the cert header file
- ASN_OBJECT_ID_E Returned if there is an error parsing the encryption type from the cert
- ASN_EXPECT_0_E Returned if there is a formatting error in the encryption specification of the cert file

- ASN_BEFORE_DATE_E Returned if the date is before the certificate start date
- ASN_AFTER_DATE_E Returned if the date is after the certificate expiration date
- ASN_BITSTR_E Returned if there is an error parsing a bit string from the certificate
- ECC_CURVE_OID_E Returned if there is an error parsing the ECC key from the certificate
- ASN_UNKNOWN_OID_E Returned if the certificate is using an unknown key object id
- ASN_VERSION_E Returned if the ALLOW_V1_EXTENSIONS option is not defined and the certificate is a V1 or V2 certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension
- ASN_CRIT_EXT_E Returned if an unfamiliar critical extension is encountered in processing the certificate
- ASN_SIG_OID_E Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
- ASN_SIG_CONFIRM_E Returned if confirming the certification signature fails
- ASN_NAME_INVALID_E Returned if the certificate's name is not permitted by the CA name constraints
- ASN_NO_SIGNER_E Returned if there is no CA signer to verify the certificate's authenticity

Example

```
Cert myCert;
// initialize myCert
if(wc_SetSubject(&myCert, "./path/to/ca-cert.pem") != 0) {
    // error setting subject
}
```

```
int wc_SetSubjectRaw(
    Cert * cert,
    const byte * der,
    int derSz
)
```

This function sets the raw subject for a certificate from the subject in the provided der buffer. This method is used to set the raw subject field prior to signing.

Parameters:

- **cert** pointer to the cert for which to set the raw subject
- **der** pointer to the buffer containing the der formatted certificate from which to grab the subject
- **derSz** size of the buffer containing the der formatted certificate from which to grab the subject

See:

- wc_InitCert
- wc_SetSubject

Return:

- 0 Returned on successfully setting the subject for the certificate
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_PARSE_E Returned if there is an error parsing the cert header file

- ASN_OBJECT_ID_E Returned if there is an error parsing the encryption type from the cert
- ASN_EXPECT_0_E Returned if there is a formatting error in the encryption specification of the cert file
- ASN_BEFORE_DATE_E Returned if the date is before the certificate start date
- ASN_AFTER_DATE_E Returned if the date is after the certificate expiration date
- ASN_BITSTR_E Returned if there is an error parsing a bit string from the certificate
- ECC_CURVE_OID_E Returned if there is an error parsing the ECC key from the certificate
- ASN_UNKNOWN_OID_E Returned if the certificate is using an unknown key object id
- ASN_VERSION_E Returned if the ALLOW_V1_EXTENSIONS option is not defined and the certificate is a V1 or V2 certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension
- ASN_CRIT_EXT_E Returned if an unfamiliar critical extension is encountered in processing the certificate
- ASN_SIG_OID_E Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
- ASN_SIG_CONFIRM_E Returned if confirming the certification signature fails
- ASN_NAME_INVALID_E Returned if the certificate's name is not permitted by the CA name constraints
- ASN_NO_SIGNER_E Returned if there is no CA signer to verify the certificate's authenticity

Example

```
Cert myCert;
// initialize myCert
byte* der;
der = (byte*)malloc(FOURK_BUF);
// initialize der
if(wc_SetSubjectRaw(&myCert, der, FOURK_BUF) != 0) {
    // error setting subject
}
```

```
int wc_GetSubjectRaw(
    byte ** subjectRaw,
    Cert * cert
)
```

This function gets the raw subject from the certificate structure.

Parameters:

- **subjectRaw** pointer-pointer to the raw subject upon successful return
- **cert** pointer to the cert from which to get the raw subject

See:

- wc_InitCert
- wc_SetSubjectRaw

Return:

- 0 Returned on successfully getting the subject from the certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension

Example

```
Cert myCert;
byte *subjRaw;
// initialize myCert

if(wc_GetSubjectRaw(&subjRaw, &myCert) != 0) {
    // error setting subject
}
```

```
int wc_SetAltNames(
    Cert * cert,
    const char * file
)
```

This function sets the alternate names for a certificate to the alternate names in the provided pem file. This is useful in the case that one wishes to secure multiple domains with the same certificate. This method is used to set fields prior to signing.

Parameters:

- **cert** pointer to the cert for which to set the alt names
- **file** path of the file containing the pem formatted certificate

See:

- wc_InitCert
- wc_SetIssuer

Return:

- 0 Returned on successfully setting the alt names for the certificate
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_PARSE_E Returned if there is an error parsing the cert header file
- ASN_OBJECT_ID_E Returned if there is an error parsing the encryption type from the cert
- ASN_EXPECT_0_E Returned if there is a formatting error in the encryption specification of the cert file
- ASN_BEFORE_DATE_E Returned if the date is before the certificate start date
- ASN_AFTER_DATE_E Returned if the date is after the certificate expiration date
- ASN_BITSTR_E Returned if there is an error parsing a bit string from the certificate
- ECC_CURVE_OID_E Returned if there is an error parsing the ECC key from the certificate
- ASN_UNKNOWN_OID_E Returned if the certificate is using an unknown key object id
- ASN_VERSION_E Returned if the ALLOW_V1_EXTENSIONS option is not defined and the certificate is a V1 or V2 certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension
- ASN_CRIT_EXT_E Returned if an unfamiliar critical extension is encountered in processing the certificate

- ASN_SIG_OID_E Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
- ASN_SIG_CONFIRM_E Returned if confirming the certification signature fails
- ASN_NAME_INVALID_E Returned if the certificate's name is not permitted by the CA name constraints
- ASN_NO_SIGNER_E Returned if there is no CA signer to verify the certificate's authenticity

Example

```
Cert myCert;
// initialize myCert
if(wc_SetSubject(&myCert, "./path/to/ca-cert.pem") != 0) {
    // error setting alt names
}
```

```
int wc_SetIssuerBuffer(
    Cert * cert,
    const byte * der,
    int derSz
)
```

This function sets the issuer for a certificate from the issuer in the provided der buffer. It also changes the certificate's self-signed attribute to false. This method is used to set fields prior to signing.

Parameters:

- **cert** pointer to the cert for which to set the issuer
- **der** pointer to the buffer containing the der formatted certificate from which to grab the issuer
- **derSz** size of the buffer containing the der formatted certificate from which to grab the issuer

See:

- wc_InitCert
- wc_SetIssuer

Return:

- 0 Returned on successfully setting the issuer for the certificate
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_PARSE_E Returned if there is an error parsing the cert header file
- ASN_OBJECT_ID_E Returned if there is an error parsing the encryption type from the cert
- ASN_EXPECT_0_E Returned if there is a formatting error in the encryption specification of the cert file
- ASN_BEFORE_DATE_E Returned if the date is before the certificate start date
- ASN_AFTER_DATE_E Returned if the date is after the certificate expiration date
- ASN_BITSTR_E Returned if there is an error parsing a bit string from the certificate
- ECC_CURVE_OID_E Returned if there is an error parsing the ECC key from the certificate
- ASN_UNKNOWN_OID_E Returned if the certificate is using an unknown key object id
- ASN_VERSION_E Returned if the ALLOW_V1_EXTENSIONS option is not defined and the certificate is a V1 or V2 certificate

- BAD_FUNC_ARG Returned if there is an error processing the certificate extension
- ASN_CRIT_EXT_E Returned if an unfamiliar critical extension is encountered in processing the certificate
- ASN_SIG_OID_E Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
- ASN_SIG_CONFIRM_E Returned if confirming the certification signature fails
- ASN_NAME_INVALID_E Returned if the certificate's name is not permitted by the CA name constraints
- ASN_NO_SIGNER_E Returned if there is no CA signer to verify the certificate's authenticity

Example

```
Cert myCert;
// initialize myCert
byte* der;
der = (byte*)malloc(FOURK_BUF);
// initialize der
if(wc_SetIssuerBuffer(&myCert, der, FOURK_BUF) != 0) {
    // error setting issuer
}
```

```
int wc_SetIssuerRaw(
    Cert * cert,
    const byte * der,
    int derSz
)
```

This function sets the raw issuer for a certificate from the issuer in the provided der buffer. This method is used to set the raw issuer field prior to signing.

Parameters:

- **cert** pointer to the cert for which to set the raw issuer
- **der** pointer to the buffer containing the der formatted certificate from which to grab the subject
- **derSz** size of the buffer containing the der formatted certificate from which to grab the subject

See:

- wc_InitCert
- wc_SetIssuer

Return:

- 0 Returned on successfully setting the issuer for the certificate
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_PARSE_E Returned if there is an error parsing the cert header file
- ASN_OBJECT_ID_E Returned if there is an error parsing the encryption type from the cert
- ASN_EXPECT_0_E Returned if there is a formatting error in the encryption specification of the cert file
- ASN_BEFORE_DATE_E Returned if the date is before the certificate start date

- ASN_AFTER_DATE_E Returned if the date is after the certificate expiration date
- ASN_BITSTR_E Returned if there is an error parsing a bit string from the certificate
- ECC_CURVE_OID_E Returned if there is an error parsing the ECC key from the certificate
- ASN_UNKNOWN_OID_E Returned if the certificate is using an unknown key object id
- ASN_VERSION_E Returned if the ALLOW_V1_EXTENSIONS option is not defined and the certificate is a V1 or V2 certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension
- ASN_CRIT_EXT_E Returned if an unfamiliar critical extension is encountered in processing the certificate
- ASN_SIG_OID_E Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
- ASN_SIG_CONFIRM_E Returned if confirming the certification signature fails
- ASN_NAME_INVALID_E Returned if the certificate's name is not permitted by the CA name constraints
- ASN_NO_SIGNER_E Returned if there is no CA signer to verify the certificate's authenticity

Example

```
Cert myCert;
// initialize myCert
byte* der;
der = (byte*)malloc(FOURK_BUF);
// initialize der
if(wc_SetIssuerRaw(&myCert, der, FOURK_BUF) != 0) {
    // error setting subject
}
```

```
int wc_SetSubjectBuffer(
    Cert * cert,
    const byte * der,
    int derSz
)
```

This function sets the subject for a certificate from the subject in the provided der buffer. This method is used to set fields prior to signing.

Parameters:

- **cert** pointer to the cert for which to set the subject
- **der** pointer to the buffer containing the der formatted certificate from which to grab the subject
- **derSz** size of the buffer containing the der formatted certificate from which to grab the subject

See:

- wc_InitCert
- wc_SetSubject

Return:

- 0 Returned on successfully setting the subject for the certificate

- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_PARSE_E Returned if there is an error parsing the cert header file
- ASN_OBJECT_ID_E Returned if there is an error parsing the encryption type from the cert
- ASN_EXPECT_0_E Returned if there is a formatting error in the encryption specification of the cert file
- ASN_BEFORE_DATE_E Returned if the date is before the certificate start date
- ASN_AFTER_DATE_E Returned if the date is after the certificate expiration date
- ASN_BITSTR_E Returned if there is an error parsing a bit string from the certificate
- ECC_CURVE_OID_E Returned if there is an error parsing the ECC key from the certificate
- ASN_UNKNOWN_OID_E Returned if the certificate is using an unknown key object id
- ASN_VERSION_E Returned if the ALLOW_V1_EXTENSIONS option is not defined and the certificate is a V1 or V2 certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension
- ASN_CRIT_EXT_E Returned if an unfamiliar critical extension is encountered in processing the certificate
- ASN_SIG_OID_E Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
- ASN_SIG_CONFIRM_E Returned if confirming the certification signature fails
- ASN_NAME_INVALID_E Returned if the certificate's name is not permitted by the CA name constraints
- ASN_NO_SIGNER_E Returned if there is no CA signer to verify the certificate's authenticity

Example

```
Cert myCert;
// initialize myCert
byte* der;
der = (byte*)malloc(FOURK_BUF);
// initialize der
if(wc_SetSubjectBuffer(&myCert, der, FOURK_BUF) != 0) {
    // error setting subject
}
```

```
int wc_SetAltNamesBuffer(
    Cert * cert,
    const byte * der,
    int derSz
)
```

This function sets the alternate names for a certificate from the alternate names in the provided der buffer. This is useful in the case that one wishes to secure multiple domains with the same certificate. This method is used to set fields prior to signing.

Parameters:

- **cert** pointer to the cert for which to set the alternate names
- **der** pointer to the buffer containing the der formatted certificate from which to grab the alternate names
- **derSz** size of the buffer containing the der formatted certificate from which to grab the alternate names

See:

- wc_InitCert
- wc_SetAltNames

Return:

- 0 Returned on successfully setting the alternate names for the certificate
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_PARSE_E Returned if there is an error parsing the cert header file
- ASN_OBJECT_ID_E Returned if there is an error parsing the encryption type from the cert
- ASN_EXPECT_0_E Returned if there is a formatting error in the encryption specification of the cert file
- ASN_BEFORE_DATE_E Returned if the date is before the certificate start date
- ASN_AFTER_DATE_E Returned if the date is after the certificate expiration date
- ASN_BITSTR_E Returned if there is an error parsing a bit string from the certificate
- ECC_CURVE_OID_E Returned if there is an error parsing the ECC key from the certificate
- ASN_UNKNOWN_OID_E Returned if the certificate is using an unknown key object id
- ASN_VERSION_E Returned if the ALLOW_V1_EXTENSIONS option is not defined and the certificate is a V1 or V2 certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension
- ASN_CRIT_EXT_E Returned if an unfamiliar critical extension is encountered in processing the certificate
- ASN_SIG_OID_E Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
- ASN_SIG_CONFIRM_E Returned if confirming the certification signature fails
- ASN_NAME_INVALID_E Returned if the certificate's name is not permitted by the CA name constraints
- ASN_NO_SIGNER_E Returned if there is no CA signer to verify the certificate's authenticity

Example

```
Cert myCert;
// initialize myCert
byte* der;
der = (byte*)malloc(FOURK_BUF);
// initialize der
if(wc_SetAltNamesBuffer(&myCert, der, FOURK_BUF) != 0) {
    // error setting subject
}
```

```
int wc_SetDNSEntry(
    void * heap,
    const char * str,
    int strLen,
    int type,
    DNS_entry ** entries
)
```

This function allocates a single subject alternative name (SAN) entry, copies the supplied name into it, sets its GeneralName type and length, and appends it to the linked list pointed to by entries. The

name is duplicated internally, so the caller's buffer need not outlive the call. The resulting list can be encoded with wc_FlattenAltNames and must be freed with FreeAltNames.

Parameters:

- **heap** pointer to the heap hint used for allocations (may be NULL)
- **str** pointer to the name bytes (e.g. a DNS string, or raw IP octets for ASN_IP_TYPE)
- **strLen** length of str in bytes
- **type** GeneralName type (e.g. ASN_DNS_TYPE, ASN_IP_TYPE, ASN_RFC822_TYPE, ASN_URI_TYPE)
- **entries** in/out pointer to the head of the alt-name linked list; a new entry is appended

See:

- wc_FlattenAltNames
- FreeAltNames

Return:

- 0 Returned on success.
- MEMORY_E Returned if dynamic memory allocation fails.
- BAD_FUNC_ARG Returned if str or entries is NULL, strLen is negative, or type is not a supported GeneralName type; also returned for an ASN_IP_TYPE entry whose length is not a valid IPv4/IPv6 address, or an ASN_RID_TYPE entry with malformed contents.
- BUFFER_E Returned if the string representation of an ASN_IP_TYPE or ASN_RID_TYPE entry does not fit its internal buffer.
- Other negative error codes may propagate from generating the string form of ASN_IP_TYPE and ASN_RID_TYPE entries.

Note:

- This helper (along with wc_FlattenAltNames and FreeAltNames) is exported from the library only when WOLFSSL_PUBLIC_ASN, OPENSSL_EXTRA, OPENSSL_EXTRA_X509_SMALL, or WOLFSSL_TEST_CERT is defined; its prototype lives in wolfssl/wolfcrypt/asn.h (not asn_public.h) because it uses the DNS_entry type.
- This function additionally requires WOLFSSL_ASN_TEMPLATE (its internal SetDNSEntry/AddDNSEntryToList helpers are template-only), on top of the WOLFSSL_CERT_GEN && WOLFSSL_ALT_NAMES gating shared with its companions wc_FlattenAltNames and wc_SetAltNamesFromList, which do not require WOLFSSL_ASN_TEMPLATE. Because this builder is the more restrictive of the set, a DNS_entry list built here can always be encoded by a public API in the same build; in non-template builds the public list can only be sourced from a parsed DecodedCert.altNames.

Example

```
DNS_entry* list = NULL;
if (wc_SetDNSEntry(NULL, "example.com", 11, ASN_DNS_TYPE, &list) != 0) {
    // error adding alt name
}
// ... encode with wc_FlattenAltNames, then:
FreeAltNames(list, NULL);
```

```
int wc_FlattenAltNames(
    byte * output,
    word32 outputSz,
    const DNS_entry * names
)
```

This function encodes a linked list of subject alternative name entries into the DER GeneralNames SEQUENCE used as the value of the subjectAltName certificate extension. The output is suitable for assigning to Cert.altNames (with the return value stored in Cert.altNamesSz) prior to signing.

Parameters:

- **output** buffer that receives the DER GeneralNames SEQUENCE; size it to hold the full extension value (e.g. CTC_MAX_ALT_SIZE)
- **outputSz** capacity of output in bytes
- **names** head of the alt-name linked list to encode (e.g. built with wc_SetDNSEntry, or taken from a parsed DecodedCert.altNames)

See:

- wc_SetDNSEntry
- wc_SetAltNamesFromList
- FreeAltNames
- wc_SetAltNamesBuffer

Return:

- 0 the number of bytes written to output (the full SEQUENCE, including its tag and length).
- 0 Returned when names is NULL (nothing to encode).
- BAD_FUNC_ARG Returned if output is NULL.
- BUFFER_E Returned if output is too small to hold the encoding.

Note: This helper (along with wc_SetDNSEntry and FreeAltNames) is exported from the library only when WOLFSSL_PUBLIC_ASN, OPENSSL_EXTRA, OPENSSL_EXTRA_X509_SMALL, or WOLFSSL_TEST_CERT is defined; its prototype lives in wolfssl/wolfcrypt/asn.h (not asn_public.h) because it uses the DNS_entry type.

Example

```
Cert cert;
DNS_entry* list = NULL;
// ... populate list with wc_SetDNSEntry ...
int n = wc_FlattenAltNames(cert.altNames, sizeof(cert.altNames), list);
if (n < 0) {
    // error encoding alt names
}
cert.altNamesSz = n;
FreeAltNames(list, NULL);
```

```
int wc_SetAltNamesFromList(
    Cert * cert,
    const DNS_entry * names
)
```

This function encodes a linked list of subject alternative name entries directly into a Cert structure, ready for signing. It is a convenience wrapper around wc_FlattenAltNames: the list is encoded into cert->altNames and the encoded length is stored in cert->altNamesSz, so the caller does not have to manage the buffer or size bookkeeping. The supplied list is not consumed and must still be freed by the caller with FreeAltNames.

Parameters:

- **cert** pointer to the Cert whose altNames/altNamesSz fields are set
- **names** head of the alt-name linked list to encode (e.g. built with wc_SetDNSEntry, or taken from a parsed DecodedCert.altNames); may be NULL, in which case cert->altNamesSz is set to 0

See:

- wc_SetDNSEntry
- wc_FlattenAltNames
- wc_SetAltNamesBuffer
- FreeAltNames

Return:

- 0 Returned on success.
- BAD_FUNC_ARG Returned if cert is NULL.
- BUFFER_E Returned if the encoded names do not fit in cert->altNames.

Note: This helper (along with wc_SetDNSEntry, wc_FlattenAltNames, and FreeAltNames) is exported from the library only when WOLFSSL_PUBLIC_ASN, OPENSSL_EXTRA, OPENSSL_EXTRA_X509_SMALL, or WOLFSSL_TEST_CERT is defined; its prototype lives in wolfssl/wolfcrypt/asn.h (not asn_public.h) because it uses the DNS_entry type. Its DER-input sibling wc_SetAltNamesBuffer is always exported.

Example

```
Cert cert;
DNS_entry* list = NULL;
wc_InitCert(&cert);
// ... populate list with wc_SetDNSEntry ...
if (wc_SetAltNamesFromList(&cert, list) != 0) {
    // error encoding alt names
}
FreeAltNames(list, NULL);
// ... wc_MakeCert / wc_SignCert ...
```

```
int wc_SetDatesBuffer(
    Cert * cert,
    const byte * der,
    int derSz
)
```

This function sets the dates for a certificate from the date range in the provided der buffer. This method is used to set fields prior to signing.

Parameters:

- **cert** pointer to the cert for which to set the dates
- **der** pointer to the buffer containing the der formatted certificate from which to grab the date range
- **derSz** size of the buffer containing the der formatted certificate from which to grab the date range

See: wc_InitCert

Return:

- 0 Returned on successfully setting the dates for the certificate
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_PARSE_E Returned if there is an error parsing the cert header file
- ASN_OBJECT_ID_E Returned if there is an error parsing the encryption type from the cert
- ASN_EXPECT_0_E Returned if there is a formatting error in the encryption specification of the cert file
- ASN_BEFORE_DATE_E Returned if the date is before the certificate start date
- ASN_AFTER_DATE_E Returned if the date is after the certificate expiration date
- ASN_BITSTR_E Returned if there is an error parsing a bit string from the certificate
- ECC_CURVE_OID_E Returned if there is an error parsing the ECC key from the certificate
- ASN_UNKNOWN_OID_E Returned if the certificate is using an unknown key object id
- ASN_VERSION_E Returned if the ALLOW_V1_EXTENSIONS option is not defined and the certificate is a V1 or V2 certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension
- ASN_CRIT_EXT_E Returned if an unfamiliar critical extension is encountered in processing the certificate
- ASN_SIG_OID_E Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
- ASN_SIG_CONFIRM_E Returned if confirming the certification signature fails
- ASN_NAME_INVALID_E Returned if the certificate's name is not permitted by the CA name constraints
- ASN_NO_SIGNER_E Returned if there is no CA signer to verify the certificate's authenticity

Example

```
Cert myCert;
// initialize myCert
byte* der;
der = (byte*)malloc(FOURK_BUF);
// initialize der
if(wc_SetDatesBuffer(&myCert, der, FOURK_BUF) != 0) {
```

```
    // error setting subject
}



int wc_SetAuthKeyIdFromPublicKey(
    Cert * cert,
    RsaKey * rsakey,
    ecc_key * eckey
)
```

Set AKID from either an RSA or ECC public key. note: Only set one of rsakey or eckey, not both.

Parameters:

- **cert** Pointer to the certificate to set the SKID.
- **rsakey** Pointer to the RsaKey struct to read from.
- **eckey** Pointer to the ecc_key to read from.

See:

- wc_SetSubjectKeyId
- wc_SetAuthKeyId
- wc_SetAuthKeyIdFromCert

Return:

- 0 Success
- BAD_FUNC_ARG Either cert is null or both rsakey and eckey are null.
- MEMORY_E Error allocating memory.
- PUBLIC_KEY_E Error writing to the key.

Example

```
Cert myCert;
RsaKey keypub;

wc_InitRsaKey(&keypub, 0);

if (wc_SetAuthKeyIdFromPublicKey(&myCert, &keypub, NULL) != 0)
{
    // Handle error
}



int wc_SetAuthKeyIdFromPublicKey_ex(
    Cert * cert,
    int keyType,
    void * key
)
```

Sets authority key ID from public key with generic key type.

Parameters:

- **cert** Certificate structure
- **keyType** Key type (RSA_TYPE, ECC_TYPE, ED25519_TYPE, etc.)
- **key** Pointer to key structure

See: wc_SetAuthKeyIdFromPublicKey

Return:

- 0 on success
- BAD_FUNC_ARG if parameters invalid
- MEMORY_E if memory allocation fails

Example

```
Cert myCert;
wc_InitCert(&myCert);
RsaKey key;
int ret = wc_SetAuthKeyIdFromPublicKey_ex(&myCert, RSA_TYPE,
                                          &key);
```

```
int wc_SetAuthKeyIdFromCert(
    Cert * cert,
    const byte * der,
    int derSz
)
```

Set AKID from from DER encoded certificate.

Parameters:

- **cert** The Cert struct to write to.
- **der** The DER encoded certificate buffer.
- **derSz** Size of der in bytes.

See:

- wc_SetAuthKeyIdFromPublicKey
- wc_SetAuthKeyId

Return:

- 0 Success
- BAD_FUNC_ARG Error if any argument is null or derSz is less than 0.
- MEMORY_E Error if problem allocating memory.
- ASN_NO_SKID No subject key ID found.

Example

```
Cert some_cert;
byte some_der[] = { // Initialize a DER buffer };
wc_InitCert(&some_cert);
if(wc_SetAuthKeyIdFromCert(&some_cert, some_der, sizeof(some_der) != 0)
{
    // Handle error
}
```

```
int wc_SetAuthKeyId(
    Cert * cert,
    const char * file
)
```

Set AKID from certificate file in PEM format.

Parameters:

- **cert** Cert struct you want to set the AKID of.
- **file** Buffer containing PEM cert file.

See:

- wc_SetAuthKeyIdFromPublicKey
- wc_SetAuthKeyIdFromCert

Return:

- 0 Success
- BAD_FUNC_ARG Error if cert or file is null.
- MEMORY_E Error if problem allocating memory.

Example

```
char* file_name = "/path/to/file";
cert some_cert;
wc_InitCert(&some_cert);

if(wc_SetAuthKeyId(&some_cert, file_name) != 0)
{
    // Handle Error
}
```

```
int wc_SetSubjectKeyIdFromPublicKey(
    Cert * cert,
    RsaKey * rsakey,
```

```
    ecc_key * eckey
)
```

Set SKID from RSA or ECC public key.

Parameters:

- **cert** Pointer to a Cert structure to be used.
- **rsakey** Pointer to an RsaKey structure
- **eckey** Pointer to an ecc_key structure

See: wc_SetSubjectKeyId

Return:

- 0 Success
- BAD_FUNC_ARG Returned if cert or rsakey and eckey are null.
- MEMORY_E Returned if there is an error allocating memory.
- PUBLIC_KEY_E Returned if there is an error getting the public key.

Example

```
Cert some_cert;
RsaKey some_key;
wc_InitCert(&some_cert);
wc_InitRsaKey(&some_key);

if(wc_SetSubjectKeyIdFromPublicKey(&some_cert,&some_key, NULL) != 0)
{
    // Handle Error
}
```

```
int wc_SetSubjectKeyIdFromPublicKey_ex(
    Cert * cert,
    int keyType,
    void * key
)
```

Sets subject key ID from public key with generic key type.

Parameters:

- **cert** Certificate structure
- **keyType** Key type (RSA_TYPE, ECC_TYPE, ED25519_TYPE, etc.)
- **key** Pointer to key structure

See: wc_SetSubjectKeyIdFromPublicKey

Return:

- 0 on success

- BAD_FUNC_ARG if parameters invalid
- MEMORY_E if memory allocation fails
- PUBLIC_KEY_E if error getting public key

Example

```
Cert myCert;
wc_InitCert(&myCert);
EccKey key;
int ret = wc_SetSubjectKeyIdFromPublicKey_ex(&myCert, ECC_TYPE,
                                             &key);
```

```
int wc_SetSubjectKeyId(
    Cert * cert,
    const char * file
)
```

Set SKID from public key file in PEM format. Both arguments are required.

Parameters:

- **cert** Cert structure to set the SKID of.
- **file** Contains the PEM encoded file.

See: wc_SetSubjectKeyIdFromPublicKey

Return:

- 0 Success
- BAD_FUNC_ARG Returns if cert or file is null.
- MEMORY_E Returns if there is a problem allocating memory for key.
- PUBLIC_KEY_E Returns if there is an error decoding the public key.

Example

```
const char* file_name = "path/to/file";
Cert some_cert;
wc_InitCert(&some_cert);

if(wc_SetSubjectKeyId(&some_cert, file_name) != 0)
{
    // Handle Error
}
```

```
int wc_SetExtKeyUsage(
    Cert * cert,
    const char * value
)
```

Sets extended key usage using comma-delimited string.

Parameters:

- **cert** Certificate structure
- **value** Comma-delimited string of extended key usage values

See:

- wc_SetKeyUsage
- wc_SetExtKeyUsageOID

Return:

- 0 on success
- BAD_FUNC_ARG if parameters invalid
- MEMORY_E if memory allocation fails

Example

```
Cert myCert;
wc_InitCert(&myCert);
int ret = wc_SetExtKeyUsage(&myCert,
                            "serverAuth,clientAuth");
```

```
int wc_SetExtKeyUsageOID(
    Cert * cert,
    const char * oid,
    word32 sz,
    byte idx,
    void * heap
)
```

Sets extended key usage using OID string.

Parameters:

- **cert** Certificate structure
- **oid** OID string
- **sz** Length of OID string
- **idx** Index for multiple OIDs
- **heap** Heap hint for memory allocation

See: wc_SetExtKeyUsage

Return:

- 0 on success
- BAD_FUNC_ARG if parameters invalid
- MEMORY_E if memory allocation fails

Example

```
Cert myCert;
wc_InitCert(&myCert);
const char* oid = "1.3.6.1.5.5.7.3.1";
int ret = wc_SetExtKeyUsageOID(&myCert, oid, strlen(oid),
                               0, NULL);
```

```
int wc_PemPubKeyToDer(
    const char * fileName,
    unsigned char * derBuf,
    int derSz
)
```

Loads a PEM key from a file and converts to a DER encoded buffer.

Parameters:

- **fileName** Name of the file to load.
- **derBuf** Buffer for DER encoded key.
- **derSz** Size of DER buffer.

See: wc_PubKeyPemToDer

Return:

- 0 Success
- <0 Error
- SSL_BAD_FILE There is a problem with opening the file.
- MEMORY_E There is an error allocating memory for the file buffer.
- BUFFER_E derBuf is not large enough to hold the converted key.

Example

```
char* some_file = "filename";
unsigned char der[];

if(wc_PemPubKeyToDer(some_file, der, sizeof(der)) != 0)
{
    //Handle Error
}
```

```
int wc_PemPubKeyToDer_ex(
    const char * fileName,
    DerBuffer ** der
)
```

Loads PEM public key from file to DER buffer.

Parameters:

- **fileName** Path to PEM file
- **der** Pointer to DerBuffer pointer to allocate

See:

- wc_PemPubKeyToDer
- wc_FreeDer

Return:

- 0 on success
- negative on error

Example

```
DerBuffer* der = NULL;
int ret = wc_PemPubKeyToDer_ex("pubkey.pem", &der);
if (ret == 0) {
    // Use der->buffer and der->length
    wc_FreeDer(&der);
}
```

```
int wc_PubKeyPemToDer(
    const unsigned char * pem,
    int pemSz,
    unsigned char * buff,
    int buffSz
)
```

Convert a PEM encoded public key to DER. Returns the number of bytes written to the buffer or a negative value for an error.

Parameters:

- **pem** PEM encoded key
- **pemSz** Size of pem
- **buff** Pointer to buffer for output.
- **buffSz** Size of buffer.

See: wc_PemPubKeyToDer

Return:

- 0 Success, number of bytes written.

- BAD_FUNC_ARG Returns if pem, buff, or buffSz are null
- <0 An error occurred in the function.

Example

```
byte some_pem[] = { Initialize with PEM key }
unsigned char out_buffer[1024]; // Ensure buffer is large enough to fit DER

if(wc_PubKeyPemToDer(some_pem, sizeof(some_pem), out_buffer,
sizeof(out_buffer)) < 0)
{
    // Handle error
}
```

```
int wc_PemGetHeaderFooter(
    int type,
    const char ** header,
    const char ** footer
)
```

Gets PEM header and footer strings for given type.

Parameters:

- **type** PEM type (CERT_TYPE, PRIVATEKEY_TYPE, etc.)
- **header** Pointer to header string pointer
- **footer** Pointer to footer string pointer

See: wc_PemToDer

Return:

- 0 on success
- BAD_FUNC_ARG if parameters invalid

Example

```
const char* header;
const char* footer;
int ret = wc_PemGetHeaderFooter(CERT_TYPE, &header, &footer);
```

```
int wc_AllocDer(
    DerBuffer ** pDer,
    word32 length,
    int type,
    void * heap
)
```

Allocates DER buffer with specified length and type.

Parameters:

- **pDer** Pointer to DerBuffer pointer to allocate
- **length** Length of buffer to allocate
- **type** Buffer type for tracking
- **heap** Heap hint for memory allocation

See: wc_FreeDer

Return:

- 0 on success
- BAD_FUNC_ARG if pDer is NULL
- MEMORY_E if allocation fails

Example

```
DerBuffer* der = NULL;
int ret = wc_AllocDer(&der, 1024, CERT_TYPE, NULL);
if (ret == 0) {
    // Use der->buffer
    wc_FreeDer(&der);
}
```

```
void wc_FreeDer(
    DerBuffer ** pDer
)
```

Frees DER buffer allocated by wc_AllocDer or wc_PemToDer.

Parameters:

- **pDer** Pointer to DerBuffer pointer to free

See:

- wc_AllocDer
- wc_PemToDer

Example

```
DerBuffer* der = NULL;
wc_AllocDer(&der, 1024, CERT_TYPE, NULL);
// Use der
wc_FreeDer(&der);
```

```
int wc_PemToDer(
    const unsigned char * buff,
    long longSz,
    int type,
    DerBuffer ** pDer,
    void * heap,
    EncryptedInfo * info,
    int * keyFormat
)
```

Converts PEM to DER format with encryption info support.

Parameters:

- **buff** PEM buffer
- **longSz** Size of PEM buffer
- **type** PEM type (CERT_TYPE, PRIVATEKEY_TYPE, etc.)
- **pDer** Pointer to DerBuffer pointer to allocate
- **heap** Heap hint for memory allocation
- **info** Encryption info for encrypted PEM
- **keyFormat** Pointer to store key format

See:

- wc_PemCertToDer
- wc_FreeDer

Return:

- 0 on success
- negative on error

Example

```
const unsigned char* pem;
DerBuffer* der = NULL;
EncryptedInfo info;
int keyFormat;
int ret = wc_PemToDer(pem, pemSz, PRIVATEKEY_TYPE, &der,
                      NULL, &info, &keyFormat);
if (ret == 0) {
    wc_FreeDer(&der);
}
```

```
int wc_PemCertToDer(
    const char * fileName,
    unsigned char * derBuf,
    int derSz
)
```

This function converts a pem certificate to a der certificate, and places the resulting certificate in the derBuf buffer provided.

Parameters:

- **fileName** path to the file containing a pem certificate to convert to a der certificate
- **derBuf** pointer to a char buffer in which to store the converted certificate
- **derSz** size of the char buffer in which to store the converted certificate

See: none

Return:

- Success On success returns the size of the derBuf generated
- BUFFER_E Returned if the size of derBuf is too small to hold the certificate generated
- MEMORY_E Returned if the call to XMALLOC fails

Example

```
char * file = "./certs/client-cert.pem";
int derSz;
byte* der = (byte*)XMALLOC((8*1024), NULL, DYNAMIC_TYPE_CERT);

derSz = wc_PemCertToDer(file, der, (8*1024));
if (derSz <= 0) {
    //PemCertToDer error
}
```

```
int wc_DerToPem(
    const byte * der,
    word32 derSz,
    byte * output,
    word32 outSz,
    int type
)
```

This function converts a der formatted input certificate, contained in the der buffer, into a pem formatted output certificate, contained in the output buffer. It should be noted that this is not an in place conversion, and a separate buffer must be utilized to store the pem formatted output.

Parameters:

- **der** pointer to the buffer of the certificate to convert
- **derSz** size of the the certificate to convert
- **output** pointer to the buffer in which to store the pem formatted certificate
- **outSz** size of the buffer in which to store the pem formatted certificate
- **type** the type of certificate to generate. Valid types are: CERT_TYPE, PRIVATEKEY_TYPE, ECC_PRIVATEKEY_TYPE, and CERTREQ_TYPE.

See: wc_PemCertToDer

Return:

- Success On successfully making a pem certificate from the input der cert, returns the size of the pem cert generated.
- BAD_FUNC_ARG Returned if there is an error parsing the der file and storing it as a pem file
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_INPUT_E Returned in the case of a base64 encoding error
- BUFFER_E May be returned if the output buffer is too small to store the pem formatted certificate

Example

```
byte* der;
// initialize der with certificate
byte* pemFormatted[FOURK_BUF];

word32 pemSz;
pemSz = wc_DerToPem(der, derSz,pemFormatted,FOURK_BUF, CERT_TYPE);
```

```
int wc_DerToPemEx(
    const byte * der,
    word32 derSz,
    byte * output,
    word32 outSz,
    byte * cipher_info,
    int type
)
```

This function converts a der formatted input certificate, contained in the der buffer, into a pem formatted output certificate, contained in the output buffer. It should be noted that this is not an in place conversion, and a separate buffer must be utilized to store the pem formatted output. Allows setting cipher info.

Parameters:

- **der** pointer to the buffer of the certificate to convert
- **derSz** size of the the certificate to convert
- **output** pointer to the buffer in which to store the pem formatted certificate
- **outSz** size of the buffer in which to store the pem formatted certificate
- **cipher_info** Additional cipher information.
- **type** the type of certificate to generate. Valid types are: CERT_TYPE, PRIVATEKEY_TYPE, ECC_PRIVATEKEY_TYPE, and CERTREQ_TYPE.

See: wc_PemCertToDer

Return:

- Success On successfully making a pem certificate from the input der cert, returns the size of the pem cert generated.
- BAD_FUNC_ARG Returned if there is an error parsing the der file and storing it as a pem file
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_INPUT_E Returned in the case of a base64 encoding error
- BUFFER_E May be returned if the output buffer is too small to store the pem formatted certificate

Example

```
byte* der;
// initialize der with certificate
byte* pemFormatted[FOURK_BUF];

word32 pemSz;
byte* cipher_info[] { Additional cipher info. }
pemSz = wc_DerToPemEx(der, derSz, pemFormatted, FOURK_BUF, cipher_info,
 ↪  CERT_TYPE);
```

```
int wc_PemCertToDer_ex(
    const char * fileName,
    DerBuffer ** der
)
```

Loads PEM certificate from file to DER buffer.

Parameters:

- **fileName** Path to PEM certificate file
- **der** Pointer to DerBuffer pointer to allocate

See:

- wc_CertPemToDer
- wc_FreeDer

Return:

- 0 on success
- negative on error

Example

```
DerBuffer* der = NULL;
int ret = wc_PemCertToDer_ex("cert.pem", &der);
if (ret == 0) {
    // Use der->buffer and der->length
    wc_FreeDer(&der);
}
```

```
word32 wc_PkcsPad(
    byte * buf,
    word32 sz,
    word32 blockSz
)
```

Adds PKCS padding to buffer for RSA encryption.

Parameters:

- **buf** Buffer to pad
- **sz** Current size of data in buffer
- **blockSz** Block size for padding

See: wc_RsaPublicEncrypt

Return:

- Padded size on success
- 0 on error

Example

```
byte buffer[256];
word32 dataSz = 100;
word32 paddedSz = wc_PkcsPad(buffer, dataSz, 256);
```

```
int wc_EccPrivateKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    ecc_key * key,
    word32 inSz
)
```

This function reads in an ECC private key from the input buffer, input, parses the private key, and uses it to generate an ecc_key object, which it stores in key.

Parameters:

- **input** pointer to the buffer containing the input private key
- **inOutIdx** pointer to a word32 object containing the index in the buffer at which to start
- **key** pointer to an initialized ecc object, on which to store the decoded private key
- **inSz** size of the input buffer containing the private key

See: wc_RSA_PrivateKeyDecode

Return:

- 0 On successfully decoding the private key and storing the result in the ecc_key struct
- ASN_PARSE_E: Returned if there is an error parsing the der file and storing it as a pem file
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- BUFFER_E Returned if the certificate to convert is large than the specified max certificate size
- ASN_OBJECT_ID_E Returned if the certificate encoding has an invalid object id
- ECC_CURVE_OID_E Returned if the ECC curve of the provided key is not supported
- ECC_BAD_ARG_E Returned if there is an error in the ECC key format
- NOT_COMPILED_IN Returned if the private key is compressed, and no compression key is provided

- MP_MEM Returned if there is an error in the math library used while parsing the private key
- MP_VAL Returned if there is an error in the math library used while parsing the private key
- MP_RANGE Returned if there is an error in the math library used while parsing the private key

Example

```
int ret, idx=0;
ecc_key key; // to store key in

byte* tmp; // tmp buffer to read key from
tmp = (byte*) malloc(FOURK_BUF);

int inSz;
inSz = fread(tmp, 1, FOURK_BUF, privateKeyFile);
// read key into tmp buffer

wc_ecc_init(&key); // initialize key
ret = wc_EccPrivateKeyDecode(tmp, &idx, &key, (word32)inSz);
if(ret < 0) {
    // error decoding ecc key
}
```

```
int wc_EccKeyToDer(
    ecc_key * key,
    byte * output,
    word32 outLen
)
```

This function writes a private ECC key to der format.

Parameters:

- **key** pointer to the buffer containing the input ecc key
- **output** pointer to a buffer in which to store the der formatted key
- **outLen** the length of the buffer in which to store the der formatted key

See: wc_RsaKeyToDer

Return:

- Success On successfully writing the ECC key to der format, returns the length written to the buffer
- BAD_FUNC_ARG Returned if key or output is null, or outLen equals zero
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- BUFFER_E Returned if the converted certificate is too large to store in the output buffer
- ASN_UNKNOWN_OID_E Returned if the ECC key used is of an unknown type
- MP_MEM Returned if there is an error in the math library used while parsing the private key
- MP_VAL Returned if there is an error in the math library used while parsing the private key
- MP_RANGE Returned if there is an error in the math library used while parsing the private key

Example

```
int derSz;
ecc_key key;
// initialize and make key
byte der[FOURK_BUF];
// store der formatted key here

derSz = wc_EccKeyToDer(&key, der, FOURK_BUF);
if(derSz < 0) {
    // error converting ecc key to der buffer
}
```

```
int wc_EccPublicKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    ecc_key * key,
    word32 inSz
)
```

Decodes an ECC public key from an input buffer. It will parse an ASN sequence to retrieve the ECC key.

Parameters:

- **input** Buffer containing DER encoded key to decode.
- **inOutIdx** Index to start reading input buffer from. On output, index is set to last position parsed of input buffer.
- **key** Pointer to ecc_key struct to store the public key.
- **inSz** Size of the input buffer.

See: wc_ecc_import_x963

Return:

- 0 Success
- BAD_FUNC_ARG Returns if any arguments are null.
- ASN_PARSE_E Returns if there is an error parsing
- ASN_ECC_KEY_E Returns if there is an error importing the key. See wc_ecc_import_x963 for possible reasons.

Example

```
int ret;
word32 idx = 0;
byte buff[] = { // initialize with key };
ecc_key pubKey;
wc_ecc_init(&pubKey);
if ( wc_EccPublicKeyDecode(buff, &idx, &pubKey, sizeof(buff)) != 0) {
        // error decoding key
}
```

```
int wc_EccPublicKeyToDer(
    ecc_key * key,
    byte * output,
    word32 outLen,
    int with_AlgCurve
)
```

This function converts the ECC public key to DER format. It returns the size of buffer used. The public ECC key in DER format is stored in output buffer. The with_AlgCurve flag will include a header that has the Algorithm and Curve information.

Parameters:

- **key** Pointer to ECC key
- **output** Pointer to output buffer to write to.
- **outLen** Size of buffer.
- **with_AlgCurve** a flag for when to include a header that has the Algorithm and Curve information.

See:

- wc_EccKeyToDer
- wc_EccPrivateKeyDecode

Return:

- 0 Success, size of buffer used
- BAD_FUNC_ARG Returned if output or key is null.
- LENGTH_ONLY_E Error in getting ECC public key size.
- BUFFER_E Returned when output buffer is too small.

Example

```
ecc_key key;
wc_ecc_init(&key);
WC_RNG rng;
wc_InitRng(&rng);
wc_ecc_make_key(&rng, 32, &key);
int derSz = // Some appropriate size for der;
byte der[derSz];

if(wc_EccPublicKeyToDer(&key, der, derSz, 1) < 0)
{
    // Error converting ECC public key to der
}
```

```
int wc_EccPublicKeyToDer_ex(
    ecc_key * key,
    byte * output,
    word32 outLen,
    int with_AlgCurve,
    int comp
)
```

This function converts the ECC public key to DER format. It returns the size of buffer used. The public ECC key in DER format is stored in output buffer. The with_AlgCurve flag will include a header that has the Algorithm and Curve information. The comp parameter determines if the public key will be exported as compressed.

Parameters:

- **key** Pointer to ECC key
- **output** Pointer to output buffer to write to.
- **outLen** Size of buffer.
- **with_AlgCurve** a flag for when to include a header that has the Algorithm and Curve information.
- **comp** If 1 (non-zero) the ECC public key will be written in compressed form. If 0 it will be written in an uncompressed format.

See:

- wc_EccKeyToDer
- wc_EccPublicKeyDecode

Return:

- 0 Success, size of buffer used
- BAD_FUNC_ARG Returned if output or key is null.
- LENGTH_ONLY_E Error in getting ECC public key size.
- BUFFER_E Returned when output buffer is too small.

Example

```
ecc_key key;
wc_ecc_init(&key);
WC_RNG rng;
wc_InitRng(&rng);
wc_ecc_make_key(&rng, 32, &key);
int derSz = // Some appropriate size for der;
byte der[derSz];

// Write out a compressed ECC key
if(wc_EccPublicKeyToDer_ex(&key, der, derSz, 1, 1) < 0)
{
    // Error converting ECC public key to der
}
```

```
int wc_Curve25519PrivateKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    curve25519_key * key,
    word32 inSz
)
```

This function decodes a Curve25519 private key (only) from a DER encoded buffer.

Parameters:

- **input** Pointer to buffer containing DER encoded private key
- **inOutIdx** Index to start reading input buffer from. On output, index is set to last position parsed of input buffer.
- **key** Pointer to curve25519_key structure to store decoded key
- **inSz** Size of input DER buffer

See:

- wc_Curve25519KeyDecode
- wc_Curve25519PublicKeyDecode

Return:

- 0 Success
- BAD_FUNC_ARG Returns if input, inOutIdx or key is null
- ASN_PARSE_E Returns if there is an error parsing the DER encoded data
- ECC_BAD_ARG_E Returns if the key length is not CURVE25519_KEYSIZE or the DER key contains other issues despite being properly formatted.
- BUFFER_E Returns if the input buffer is too small to contain a valid DER encoded key.

Example

```
byte der[] = { // DER encoded key };
word32 idx = 0;
curve25519_key key;
wc_curve25519_init(&key);

if (wc_Curve25519PrivateKeyDecode(der, &idx, &key, sizeof(der)) != 0) {
    // Error decoding private key
}
```

```
int wc_Curve25519PublicKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    curve25519_key * key,
    word32 inSz
)
```

This function decodes a Curve25519 public key (only) from a DER encoded buffer.

Parameters:

- **input** Pointer to buffer containing DER encoded public key
- **inOutIdx** Index to start reading input buffer from. On output, index is set to last position parsed of input buffer.
- **key** Pointer to curve25519_key structure to store decoded key
- **inSz** Size of input DER buffer

See:

- wc_Curve25519KeyDecode
- wc_Curve25519PrivateKeyDecode

Return:

- 0 Success
- BAD_FUNC_ARG Returns if input, inOutIdx or key is null
- ASN_PARSE_E Returns if there is an error parsing the DER encoded data
- ECC_BAD_ARG_E Returns if the key length is not CURVE25519_KEYSIZE or the DER key contains other issues despite being properly formatted.
- BUFFER_E Returns if the input buffer is too small to contain a valid DER encoded key.

Example

```
byte der[] = { // DER encoded key };
word32 idx = 0;
curve25519_key key;
wc_curve25519_init(&key);
if (wc_Curve25519PublicKeyDecode(der, &idx, &key, sizeof(der)) != 0) {
    // Error decoding public key
}
```

```
int wc_Curve25519KeyDecode(
    const byte * input,
    word32 * inOutIdx,
    curve25519_key * key,
    word32 inSz
)
```

This function decodes a Curve25519 key from a DER encoded buffer. It can decode either a private key, a public key, or both.

Parameters:

- **input** Pointer to buffer containing DER encoded key
- **inOutIdx** Index to start reading input buffer from. On output, index is set to last position parsed of input buffer.
- **key** Pointer to curve25519_key structure to store decoded key

- **inSz** Size of input DER buffer

See:

- wc_Curve25519PrivateKeyDecode
- wc_Curve25519PublicKeyDecode

Return:

- 0 Success
- BAD_FUNC_ARG Returns if input, inOutIdx or key is null
- ASN_PARSE_E Returns if there is an error parsing the DER encoded data
- ECC_BAD_ARG_E Returns if the key length is not CURVE25519_KEYSIZE or the DER key contains other issues despite being properly formatted.
- BUFFER_E Returns if the input buffer is too small to contain a valid DER encoded key.

Example

```
byte der[] = { // DER encoded key };
word32 idx = 0;
curve25519_key key;
wc_curve25519_init(&key);
if (wc_Curve25519KeyDecode(der, &idx, &key, sizeof(der)) != 0) {
    // Error decoding key
}
```

```
int wc_Curve25519PrivateKeyToDer(
    curve25519_key * key,
    byte * output,
    word32 outLen
)
```

This function encodes a Curve25519 private key to DER format. If the input key structure contains a public key, it will be ignored.

Parameters:

- **key** Pointer to curve25519_key structure containing private key to encode
- **output** Buffer to hold DER encoding
- **outLen** Size of output buffer

See:

- wc_Curve25519KeyToDer
- wc_Curve25519PublicKeyToDer

Return:

-

0 Success, length of DER encoding

- BAD_FUNC_ARG Returns if key or output is null
- MEMORY_E Returns if there is an allocation failure
- BUFFER_E Returns if output buffer is too small

Example

```
curve25519_key key;
wc_curve25519_init(&key);
...
int derSz = 128; // Some appropriate size for output DER
byte der[derSz];
wc_Curve25519PrivateKeyToDer(&key, der, derSz);
```

```
int wc_Curve25519PublicKeyToDer(
    curve25519_key * key,
    byte * output,
    word32 outLen,
    int withAlg
)
```

This function encodes a Curve25519 public key to DER format. If the input key structure contains a private key, it will be ignored.

Parameters:

- **key** Pointer to curve25519_key structure containing public key to encode
- **output** Buffer to hold DER encoding
- **outLen** Size of output buffer
- **withAlg** Whether to include algorithm identifier in the DER encoding

See:

- wc_Curve25519KeyToDer
- wc_Curve25519PrivateKeyToDer

Return:

-

0 Success, length of DER encoding

- BAD_FUNC_ARG Returns if key or output is null
- MEMORY_E Returns if there is an allocation failure
- BUFFER_E Returns if output buffer is too small

Example

```
curve25519_key key;
wc_curve25519_init(&key);
...
int derSz = 128; // Some appropriate size for output DER
byte der[derSz];
wc_Curve25519PublicKeyToDer(&key, der, derSz, 1);
```

```
int wc_Curve25519KeyToDer(
    curve25519_key * key,
    byte * output,
    word32 outLen,
    int withAlg
)
```

This function encodes a Curve25519 key to DER format. It can encode either a private key, a public key, or both.

Parameters:

- **key** Pointer to curve25519_key structure containing key to encode
- **output** Buffer to hold DER encoding
- **outLen** Size of output buffer
- **withAlg** Whether to include algorithm identifier in the DER encoding

See:

- wc_Curve25519PrivateKeyToDer
- wc_Curve25519PublicKeyToDer

Return:

- 0 Success, length of DER encoding
- BAD_FUNC_ARG Returns if key or output is null
- MEMORY_E Returns if there is an allocation failure
- BUFFER_E Returns if output buffer is too small

Example

```
curve25519_key key;
wc_curve25519_init(&key);
...
int derSz = 128; // Some appropriate size for output DER
byte der[derSz];
wc_Curve25519KeyToDer(&key, der, derSz, 1);
```

```
word32 wc_EncodeSignature(
    byte * out,
    const byte * digest,
    word32 digSz,
    int hashOID
)
```

This function encodes a digital signature into the output buffer, and returns the size of the encoded signature created.

Parameters:

- **out** pointer to the buffer where the encoded signature will be written
- **digest** pointer to the digest to use to encode the signature
- **digSz** the length of the buffer containing the digest
- **hashOID** OID identifying the hash type used to generate the signature. Valid options, depending on build configurations, are: SHAh, SHA256h, SHA384h, SHA512h, MD2h, MD5h, DESb, DES3b, CTC_MD5wRSA, CTC_SHAwRSA, CTC_SHA256wRSA, CTC_SHA384wRSA, CTC_SHA512wRSA, CTC_SHAwECDSA, CTC_SHA256wECDSA, CTC_SHA384wECDSA, and CTC_SHA512wECDSA.

See: none

Return: Success On successfully writing the encoded signature to output, returns the length written to the buffer

```
int signSz;
byte encodedSig[MAX_ENCODED_SIG_SZ];
Sha256 sha256;
// initialize sha256 for hashing

byte* dig = = (byte*)malloc(WC_SHA256_DIGEST_SIZE);
// perform hashing and hash updating so dig stores SHA-256 hash
// (see wc_InitSha256, wc_Sha256Update and wc_Sha256Final)
signSz = wc_EncodeSignature(encodedSig, dig, WC_SHA256_DIGEST_SIZE, SHA256h);
```

```
int wc_GetCTC_HashOID(
    int type
)
```

This function returns the hash OID that corresponds to a hashing type. For example, when given the type: WC_SHA512, this function returns the identifier corresponding to a SHA512 hash, SHA512h.

Parameters:

- **type** the hash type for which to find the OID. Valid options, depending on build configuration, include: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512

See: none

Return:

- Success On success, returns the OID corresponding to the appropriate hash to use with that encryption type.
- 0 Returned if an unrecognized hash type is passed in as argument.

Example

```
int hashOID;

hashOID = wc_GetCTC_HashOID(WC_SHA512);
if (hashOID == 0) {
    // WOLFSSL_SHA512 not defined
}
```

```
void wc_SetCert_Free(
    Cert * cert
)
```

This function cleans up memory and resources used by the certificate structure's decoded cert cache. When WOLFSSL_CERT_GEN_CACHE is defined the decoded cert structure is cached in the certificate structure. This allows subsequent calls to certificate set functions to avoid parsing the decoded cert on each call.

Parameters:

- **cert** pointer to an uninitialized certificate information structure.

See:

- wc_SetAuthKeyIdFromCert
- wc_SetIssuerBuffer
- wc_SetSubjectBuffer
- wc_SetSubjectRaw
- wc_SetIssuerRaw
- wc_SetAltNamesBuffer
- wc_SetDatesBuffer

Return:

- 0 on success.
- BAD_FUNC_ARG Returned if invalid pointer is passed in as argument.

Example

```
Cert cert; // Initialized certificate structure

wc_SetCert_Free(&cert);
```

```
int wc_GetPkcs8TraditionalOffset(
    byte * input,
    word32 * inOutIdx,
    word32 sz
)
```

This function finds the beginning of the traditional private key inside a PKCS#8 unencrypted buffer.

Parameters:

- **input** Buffer containing unencrypted PKCS#8 private key.
- **inOutIdx** Index into the input buffer. On input, it should be a byte offset to the beginning of the the PKCS#8 buffer. On output, it will be the byte offset to the traditional private key within the input buffer.
- **sz** The number of bytes in the input buffer.

See:

- wc_CreatePKCS8Key
- wc_EncryptPKCS8Key
- wc_DecryptPKCS8Key
- wc_CreateEncryptedPKCS8Key

Return:

- Length of traditional private key on success.
- Negative values on failure.

Example

```
byte* pkcs8Buf; // Buffer containing PKCS#8 key.
word32 idx = 0;
word32 sz; // Size of pkcs8Buf.
...
ret = wc_GetPkcs8TraditionalOffset(pkcs8Buf, &idx, sz);
// pkcs8Buf + idx is now the beginning of the traditional private key bytes.
```

```
int wc_CreatePKCS8Key(
    byte * out,
    word32 * outSz,
    byte * key,
    word32 keySz,
    int algoID,
    const byte * curveOID,
    word32 oidSz
)
```

This function takes in a DER private key and converts it to PKCS#8 format. Also used in creating PKCS#12 shrouded key bags. See RFC 5208.

Parameters:

- **out** Buffer to place result in. If NULL, required out buffer size returned in outSz.
- **outSz** Size of out buffer.
- **key** Buffer with traditional DER key.
- **keySz** Size of key buffer.
- **algoID** Algorithm ID (e.g. RSAk).
- **curveOID** ECC curve OID if used. Should be NULL for RSA keys.
- **oidSz** Size of curve OID. Is set to 0 if curveOID is NULL.

See:

- wc_GetPkcs8TraditionalOffset
- wc_EncryptPKCS8Key
- wc_DecryptPKCS8Key
- wc_CreateEncryptedPKCS8Key

Return:

- The size of the PKCS#8 key placed into out on success.
- LENGTH_ONLY_E if out is NULL, with required output buffer size in outSz.
- Other negative values on failure.

Example

```
ecc_key eccKey;              // wolfSSL ECC key object.
byte* der;                   // DER-encoded ECC key.
word32 derSize;              // Size of der.
const byte* curveOid = NULL; // OID of curve used by eccKey.
word32 curveOidSz = 0;       // Size of curve OID.
byte* pkcs8;                 // Output buffer for PKCS#8 key.
word32 pkcs8Sz;              // Size of output buffer.

derSize = wc_EccKeyDerSize(&eccKey, 1);
...
derSize = wc_EccKeyToDer(&eccKey, der, derSize);
...
ret = wc_ecc_get_oid(eccKey.dp->oidSum, &curveOid, &curveOidSz);
...
ret = wc_CreatePKCS8Key(NULL, &pkcs8Sz, der,
    derSize, ECDSAk, curveOid, curveOidSz); // Get size needed in pkcs8Sz.
...
ret = wc_CreatePKCS8Key(pkcs8, &pkcs8Sz, der,
    derSize, ECDSAk, curveOid, curveOidSz);
```

```
int wc_EncryptPKCS8Key(
    byte * key,
    word32 keySz,
    byte * out,
    word32 * outSz,
    const char * password,
```

```
    int passwordSz,
    int vPKCS,
    int pbeOid,
    int encAlgId,
    byte * salt,
    word32 saltSz,
    int itt,
    WC_RNG * rng,
    void * heap
)
```

This function takes in an unencrypted PKCS#8 DER key (e.g. one created by wc_CreatePKCS8Key) and converts it to PKCS#8 encrypted format. The resulting encrypted key can be decrypted using wc_DecryptPKCS8Key. See RFC 5208.

Parameters:

- **key** Buffer with traditional DER key.
- **keySz** Size of key buffer.
- **out** Buffer to place result in. If NULL, required out buffer size returned in outSz.
- **outSz** Size of out buffer.
- **password** The password to use for the password-based encryption algorithm.
- **passwordSz** The length of the password (not including the NULL terminator).
- **vPKCS** The PKCS version to use. Can be 1 for PKCS12 or PKCS5.
- **pbeOid** The OID of the PBE scheme to use (e.g. PBES2 or one of the OIDs for PBES1 in RFC 2898 A.3).
- **encAlgId** The encryption algorithm ID to use (e.g. AES256CBCb).
- **salt** The salt buffer to use. If NULL, a random salt will be used.
- **saltSz** The length of the salt buffer. Can be 0 if passing NULL for salt.
- **itt** The number of iterations to use for the KDF.
- **rng** A pointer to an initialized WC_RNG object.
- **heap** A pointer to the heap used for dynamic allocation. Can be NULL.

See:

- wc_GetPkcs8TraditionalOffset
- wc_CreatePKCS8Key
- wc_DecryptPKCS8Key
- wc_CreateEncryptedPKCS8Key

Return:

- The size of the encrypted key placed in out on success.
- LENGTH_ONLY_E if out is NULL, with required output buffer size in outSz.
- Other negative values on failure.

Example

```
byte* pkcs8;          // Unencrypted PKCS#8 key.
word32 pkcs8Sz;       // Size of pkcs8.
byte* pkcs8Enc;       // Encrypted PKCS#8 key.
word32 pkcs8EncSz;    // Size of pkcs8Enc.
```

```
const char* password; // Password to use for encryption.
int passwordSz;       // Length of password (not including NULL terminator).
WC_RNG rng;

// The following produces an encrypted version of pkcs8 in pkcs8Enc. The
// encryption uses password-based encryption scheme 2 (PBE2) from PKCS#5 and
// the AES cipher in CBC mode with a 256-bit key. See RFC 8018 for more on
// PKCS#5.
ret = wc_EncryptPKCS8Key(pkcs8, pkcs8Sz, pkcs8Enc, &pkcs8EncSz, password,
        passwordSz, PKCS5, PBES2, AES256CBCb, NULL, 0,
        WC_PKCS12_ITT_DEFAULT, &rng, NULL);
```

```
int wc_EncryptPKCS8Key_ex(
    byte * key,
    word32 keySz,
    byte * out,
    word32 * outSz,
    const char * password,
    int passwordSz,
    int vPKCS,
    int pbeOid,
    int encAlgId,
    byte * salt,
    word32 saltSz,
    int itt,
    WC_RNG * rng,
    void * heap,
    int devId
)
```

Encrypts PKCS#8 key with extended parameters.

Parameters:

- **key** Private key buffer
- **keySz** Size of private key
- **out** Output buffer for encrypted key
- **outSz** Pointer to output buffer size (in/out)
- **password** Password for encryption
- **passwordSz** Password length
- **vPKCS** PKCS version
- **pbeOid** PBE algorithm OID
- **encAlgId** Encryption algorithm ID
- **salt** Salt buffer
- **saltSz** Salt size
- **itt** Iteration count
- **rng** Random number generator
- **heap** Heap hint for memory allocation
- **devId** Device ID for hardware acceleration

See: wc_EncryptPKCS8Key

Return:

- Size on success
- negative on error

Example

```
byte key[256], encrypted[512];
word32 encSz = sizeof(encrypted);
WC_RNG rng;
int ret = wc_EncryptPKCS8Key_ex(key, keySz, encrypted,
                                &encSz, "password", 8,
                                PKCS5, PBES2, AES256CBCb,
                                NULL, 0, 2048, &rng, NULL,
                                INVALID_DEVID);
```

```
int wc_GetTime(
    void * timePtr,
    word32 timeSize
)
```

Gets current time for certificate operations.

Parameters:

- **timePtr** Pointer to time buffer
- **timeSize** Size of time buffer

See: wc_GetDateInfo

Return:

- 0 on success
- negative on error

Example

```
time_t currentTime;
int ret = wc_GetTime(&currentTime, sizeof(currentTime));
```

```
int wc_EncryptedInfoGet(
    EncryptedInfo * info,
    const char * cipherName
)
```

Gets encryption info from encrypted PEM.

Parameters:

- **info** EncryptedInfo structure to populate
- **cipherName** Cipher name string

See: wc_PemToDer

Return:

- 0 on success
- negative on error

Example

```
EncryptedInfo info;
int ret = wc_EncryptedInfoGet(&info, "AES-256-CBC");
```

```
int wc_ParseCertPIV(
    wc_CertPIV * cert,
    const byte * buf,
    word32 totalSz
)
```

Parses PIV certificate format.

Parameters:

- **cert** PIV certificate structure to populate
- **buf** Buffer containing PIV certificate
- **totalSz** Size of buffer

See: wc_InitDecodedCert

Return:

- 0 on success
- negative on error

Example

```
wc_CertPIV cert;
int ret = wc_ParseCertPIV(&cert, pivBuf, pivSz);
```

```
int wc_GetSubjectPubKeyInfoDerFromCert(
    const byte * certDer,
    word32 certDerSz,
    byte * pubKeyDer,
    word32 * pubKeyDerSz
)
```

Extracts subject public key info from certificate.

Parameters:

- **certDer** DER encoded certificate buffer
- **certDerSz** Size of certificate
- **pubKeyDer** Output buffer for public key
- **pubKeyDerSz** Pointer to output buffer size (in/out)

See: wc_GetPubKeyDerFromCert

Return:

- Size on success
- negative on error

Example

```
byte pubKey[1024];
word32 pubKeySz = sizeof(pubKey);
int ret = wc_GetSubjectPubKeyInfoDerFromCert(certDer,
                                             certSz,
                                             pubKey,
                                             &pubKeySz);
```

```
int wc_GetDecodedCertSubject(
    const struct DecodedCert * cert,
    char * buf,
    word32 * bufSz
)
```

Retrieves the subject name from a decoded certificate.

Parameters:

- **cert** Pointer to the DecodedCert (must have been parsed).
- **buf** Output buffer to receive the subject name string, or NULL to query the required size.
- **bufSz** Pointer to the buffer size. On input, the available buffer size. On output, the number of bytes written (excluding any NUL terminator) or the required size if buf is NULL.

See:

- wc_GetDecodedCertIssuer
- wc_GetDecodedCertSerial
- wc_InitDecodedCert
- wc_ParseCert

Return:

- 0 on success.

- LENGTH_ONLY_E when buf is NULL (bufSz contains required size).
- BAD_FUNC_ARG if cert or bufSz is NULL.
- BUFFER_E if the provided buffer is too small.

This function copies the subject name string from a DecodedCert structure into the provided buffer. The string uses a one-line distinguished name format with "/" delimiters (e.g. "/C=US/O=Org/CN=example.com"). The output is NOT NUL-terminated; the caller should append a NUL byte if needed. If buf is NULL, the required buffer size is returned in bufSz and LENGTH_ONLY_E is returned.

```
int wc_GetDecodedCertIssuer(
    const struct DecodedCert * cert,
    char * buf,
    word32 * bufSz
)
```

Retrieves the issuer name from a decoded certificate.

Parameters:

- **cert** Pointer to the DecodedCert (must have been parsed).
- **buf** Output buffer to receive the issuer name string, or NULL to query the required size.
- **bufSz** Pointer to the buffer size. On input, the available buffer size. On output, the number of bytes written (excluding any NUL terminator) or the required size if buf is NULL.

See:

- wc_GetDecodedCertSubject
- wc_GetDecodedCertSerial
- wc_InitDecodedCert
- wc_ParseCert

Return:

- 0 on success.
- LENGTH_ONLY_E when buf is NULL (bufSz contains required size).
- BAD_FUNC_ARG if cert or bufSz is NULL.
- BUFFER_E if the provided buffer is too small.

This function copies the issuer name string from a DecodedCert structure into the provided buffer. The string uses a one-line distinguished name format with "/" delimiters (e.g. "/C=US/O=Org/CN=example.com"). The output is NOT NUL-terminated; the caller should append a NUL byte if needed. If buf is NULL, the required buffer size is returned in bufSz and LENGTH_ONLY_E is returned.

```
int wc_GetDecodedCertSerial(
    const struct DecodedCert * cert,
    byte * buf,
    word32 * bufSz
)
```

Retrieves the serial number from a decoded certificate.

Parameters:

- **cert** Pointer to the DecodedCert (must have been parsed).
- **buf** Output buffer to receive the serial number bytes, or NULL to query the required size.
- **bufSz** Pointer to the buffer size. On input, the available buffer size. On output, the number of bytes written or the required size if buf is NULL.

See:

- wc_GetDecodedCertSubject
- wc_GetDecodedCertIssuer
- wc_InitDecodedCert
- wc_ParseCert

Return:

- 0 on success.
- LENGTH_ONLY_E when buf is NULL (bufSz contains required size).
- BAD_FUNC_ARG if cert or bufSz is NULL.
- BUFFER_E if the provided buffer is too small.

This function copies the serial number bytes from a DecodedCert structure into the provided buffer. If buf is NULL, the required buffer size is returned in bufSz and LENGTH_ONLY_E is returned.

```
int wc_GetUUIDFromCert(
    struct DecodedCert * cert,
    byte * uuid,
    int * uuidSz
)
```

Extracts UUID from certificate.

Parameters:

- **cert** Decoded certificate structure
- **uuid** Output buffer for UUID
- **uuidSz** Pointer to UUID buffer size (in/out)

See: wc_ParseCert

Return:

- 0 on success
- negative on error

Example

```
DecodedCert cert;
byte uuid[16];
int uuidSz = sizeof(uuid);
int ret = wc_GetUUIDFromCert(&cert, uuid, &uuidSz);
```

```
int wc_GetFASCNFromCert(
    struct DecodedCert * cert,
    byte * fascn,
    int * fascnSz
)
```

Extracts FASCN from certificate.

Parameters:

- **cert** Decoded certificate structure
- **fascn** Output buffer for FASCN
- **fascnSz** Pointer to FASCN buffer size (in/out)

See: wc_ParseCert

Return:

- 0 on success
- negative on error

Example

```
DecodedCert cert;
byte fascn[25];
int fascnSz = sizeof(fascn);
int ret = wc_GetFASCNFromCert(&cert, fascn, &fascnSz);
```

```
int wc_GeneratePreTBS(
    struct DecodedCert * cert,
    byte * der,
    int derSz
)
```

Generates the pre-TBS (To Be Signed) certificate data from a decoded certificate. The TBS portion is the certificate data that gets signed by the certificate authority. This function is used in dual algorithm certificate creation where the TBS data needs to be extracted for signing with an alternative algorithm (e.g., a post-quantum algorithm).

Parameters:

- **cert** Decoded certificate structure containing the certificate to extract TBS data from
- **der** Output buffer for the pre-TBS DER-encoded data
- **derSz** Size of output buffer in bytes

See:

- wc_MakeCert
- wc_MakeSigWithBitStr

Return:

- Size of the pre-TBS data on success
- Negative error code on failure

Note: This API is only available when WOLFSSL_DUAL_ALG_CERTS is defined, which enables support for dual algorithm certificates used in Post-Quantum cryptography to provide hybrid signing with both traditional and PQ algorithms.

Example

```
DecodedCert cert;
byte preTbs[2048];
int ret = wc_GeneratePreTBS(&cert, preTbs, sizeof(preTbs));
if (ret > 0) {
    // ret contains the size of the pre-TBS data
    // preTbs can now be signed with an alternative algorithm
}
```

```
void wc_InitDecodedAcert(
    struct DecodedAcert * acert,
    void * heap
)
```

Initializes decoded attribute certificate structure.

Parameters:

- **acert** Attribute certificate structure to initialize
- **heap** Heap hint for memory allocation

See: wc_FreeDecodedAcert

Return: void

Example

```
DecodedAcert acert;
wc_InitDecodedAcert(&acert, NULL);
```

```
void wc_FreeDecodedAcert(
    struct DecodedAcert * acert
)
```

Frees decoded attribute certificate structure.

Parameters:

- **acert** Attribute certificate structure to free

See: wc_InitDecodedAcert

Return: void

Example

```
DecodedAcert acert;
wc_InitDecodedAcert(&acert, NULL);
wc_FreeDecodedAcert(&acert);
```

```
int wc_ParseX509Acert(
    struct DecodedAcert * acert,
    int verify
)
```

Parses X.509 attribute certificate.

Parameters:

- **acert** Decoded attribute certificate structure
- **verify** Non-zero to verify signature

See: wc_VerifyX509Acert

Return:

- 0 on success
- negative on error

Example

```
DecodedAcert acert;
wc_InitDecodedAcert(&acert, NULL);
int ret = wc_ParseX509Acert(&acert, 1);
```

```
int wc_VerifyX509Acert(
    const byte * acert,
    word32 acertSz,
    const byte * issuerCert,
    word32 issuerCertSz,
    void * cm
)
```

Verifies X.509 attribute certificate.

Parameters:

- **acert** Attribute certificate buffer
- **acertSz** Size of attribute certificate
- **issuerCert** Issuer certificate buffer
- **issuerCertSz** Size of issuer certificate
- **cm** Certificate manager

See: wc_ParseX509Acert

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_VerifyX509Acert(acertBuf, acertSz,
                             issuerBuf, issuerSz, cm);
```

```
int wc_DecryptPKCS8Key(
    byte * input,
    word32 sz,
    const char * password,
    int passwordSz
)
```

This function takes an encrypted PKCS#8 DER key and decrypts it to PKCS#8 unencrypted DER.Undoes the encryption done by wc_EncryptPKCS8Key. See RFC5208. The input buffer is overwritten with the decrypted data.

Parameters:

- **input** On input, buffer containing encrypted PKCS#8 key. On successful output, contains the decrypted key.
- **sz** Size of the input buffer.
- **password** The password used to encrypt the key.
- **passwordSz** The length of the password (not including NULL terminator).

See:

- wc_GetPkcs8TraditionalOffset
- wc_CreatePKCS8Key
- wc_EncryptPKCS8Key
- wc_CreateEncryptedPKCS8Key

Return:

- The length of the decrypted buffer on success.
- Negative values on failure.

Example

```
byte* pkcs8Enc;       // Encrypted PKCS#8 key made with wc_EncryptPKCS8Key.
word32 pkcs8EncSz;    // Size of pkcs8Enc.
const char* password; // Password to use for decryption.
int passwordSz;       // Length of password (not including NULL terminator).

ret = wc_DecryptPKCS8Key(pkcs8Enc, pkcs8EncSz, password, passwordSz);
```

```
int wc_CreateEncryptedPKCS8Key(
    byte * key,
    word32 keySz,
    byte * out,
    word32 * outSz,
    const char * password,
    int passwordSz,
    int vPKCS,
    int pbeOid,
    int encAlgId,
    byte * salt,
    word32 saltSz,
    int itt,
    WC_RNG * rng,
    void * heap
)
```

This function takes a traditional, DER key, converts it to PKCS#8 format, and encrypts it. It uses wc_CreatePKCS8Key and wc_EncryptPKCS8Key to do this.

Parameters:

- **key** Buffer with traditional DER key.
- **keySz** Size of key buffer.
- **out** Buffer to place result in. If NULL, required out buffer size returned in outSz.
- **outSz** Size of out buffer.
- **password** The password to use for the password-based encryption algorithm.
- **passwordSz** The length of the password (not including the NULL terminator).
- **vPKCS** The PKCS version to use. Can be 1 for PKCS12 or PKCS5.
- **pbeOid** The OID of the PBE scheme to use (e.g. PBES2 or one of the OIDs for PBES1 in RFC 2898 A.3).
- **encAlgId** The encryption algorithm ID to use (e.g. AES256CBCb).
- **salt** The salt buffer to use. If NULL, a random salt will be used.
- **saltSz** The length of the salt buffer. Can be 0 if passing NULL for salt.
- **itt** The number of iterations to use for the KDF.
- **rng** A pointer to an initialized WC_RNG object.
- **heap** A pointer to the heap used for dynamic allocation. Can be NULL.

See:

- wc_GetPkcs8TraditionalOffset
- wc_CreatePKCS8Key
- wc_EncryptPKCS8Key
- wc_DecryptPKCS8Key

Return:

- The size of the encrypted key placed in out on success.
- LENGTH_ONLY_E if out is NULL, with required output buffer size in outSz.
- Other negative values on failure.

Example

```
byte* key;            // Traditional private key (DER formatted).
word32 keySz;         // Size of key.
byte* pkcs8Enc;       // Encrypted PKCS#8 key.
word32 pkcs8EncSz;    // Size of pkcs8Enc.
const char* password; // Password to use for encryption.
int passwordSz;       // Length of password (not including NULL terminator).
WC_RNG rng;

// The following produces an encrypted, PKCS#8 version of key in pkcs8Enc.
// The encryption uses password-based encryption scheme 2 (PBE2) from PKCS#5
// and the AES cipher in CBC mode with a 256-bit key. See RFC 8018 for more
// on PKCS#5.
ret = wc_CreateEncryptedPKCS8Key(key, keySz, pkcs8Enc, &pkcs8EncSz,
        password, passwordSz, PKCS5, PBES2, AES256CBCb, NULL, 0,
        WC_PKCS12_ITT_DEFAULT, &rng, NULL);
```

```
void wc_InitDecodedCert(
    struct DecodedCert * cert,
    const byte * source,
    word32 inSz,
    void * heap
)
```

This function initializes the DecodedCert pointed to by the “cert” parameter. It saves the “source” pointer to a DER-encoded certificate of length “inSz.” This certificate can be parsed by a subsequent call to wc_ParseCert.

Parameters:

- **cert** Pointer to an allocated DecodedCert object.
- **source** Pointer to a DER-encoded certificate.
- **inSz** Length of the DER-encoded certificate in bytes.
- **heap** A pointer to the heap used for dynamic allocation. Can be NULL.

See:

- wc_ParseCert
- wc_FreeDecodedCert

Example

```
DecodedCert decodedCert; // Decoded certificate object.
byte* certBuf;           // DER-encoded certificate buffer.
word32 certBufSz;        // Size of certBuf in bytes.

wc_InitDecodedCert(&decodedCert, certBuf, certBufSz, NULL);
```

```
int wc_ParseCert(
    DecodedCert * cert,
    int type,
    int verify,
    WOLFSSL_CERT_MANAGER * cm
)
```

This function parses the DER-encoded certificate saved in the DecodedCert object and populates the fields of that object. The DecodedCert must have been initialized with a prior call to wc_InitDecodedCert. This function takes an optional pointer to a CertificateManager object, which is used to populate the certificate authority information of the DecodedCert, if the CA is found in the CertificateManager.

Parameters:

- **cert** Pointer to an initialized DecodedCert object.
- **type** Type of certificate. See the CertType enum in asn_public.h.
- **verify** Flag that, if set, indicates the user wants to verify the validity of the certificate.
- **cm** An optional pointer to a CertificateManager. Can be NULL.

See:

- wc_InitDecodedCert
- wc_FreeDecodedCert

Return:

- 0 on success.
- Other negative values on failure.

Example

```
int ret;
DecodedCert decodedCert; // Decoded certificate object.
byte* certBuf;           // DER-encoded certificate buffer.
word32 certBufSz;        // Size of certBuf in bytes.

wc_InitDecodedCert(&decodedCert, certBuf, certBufSz, NULL);
ret = wc_ParseCert(&decodedCert, CERT_TYPE, NO_VERIFY, NULL);
```

```
if (ret != 0) {
    fprintf(stderr, "wc_ParseCert failed.\n");
}
```

```
void wc_FreeDecodedCert(
    struct DecodedCert * cert
)
```

This function frees a DecodedCert that was previously initialized with wc_InitDecodedCert.

Parameters:

- **cert** Pointer to an initialized DecodedCert object.

See:

- wc_InitDecodedCert
- wc_ParseCert

Example

```
int ret;
DecodedCert decodedCert; // Decoded certificate object.
byte* certBuf;           // DER-encoded certificate buffer.
word32 certBufSz;        // Size of certBuf in bytes.

wc_InitDecodedCert(&decodedCert, certBuf, certBufSz, NULL);
ret = wc_ParseCert(&decodedCert, CERT_TYPE, NO_VERIFY, NULL);
if (ret != 0) {
    fprintf(stderr, "wc_ParseCert failed.\n");
}
wc_FreeDecodedCert(&decodedCert);
```

```
int wc_SetTimeCb(
    wc_time_cb f
)
```

This function registers a time callback that will be used anytime wolfSSL needs to get the current time. The prototype of the callback should be the same as the “time” function from the C standard library.

Parameters:

- **f** function to register as the time callback.

See: wc_Time

Return: 0 Returned on success.

Example

```
int ret = 0;
// Time callback prototype
time_t my_time_cb(time_t* t);
// Register it
ret = wc_SetTimeCb(my_time_cb);
if (ret != 0) {
    // failed to set time callback
}
time_t my_time_cb(time_t* t)
{
    // custom time function
}
```

```
time_t wc_Time(
    time_t * t
)
```

This function gets the current time. By default, it uses the XTIME macro, which varies between platforms. The user can use a function of their choosing instead via the wc_SetTimeCb function.

Parameters:

- **t** Optional time_t pointer to populate with current time.

See: wc_SetTimeCb

Return: Time Current time returned on success.

Example

```
time_t currentTime = 0;
currentTime = wc_Time(NULL);
wc_Time(&currentTime);
```

```
int wc_SetCustomExtension(
    Cert * cert,
    int critical,
    const char * oid,
    const byte * der,
    word32 derSz
)
```

This function injects a custom extension in to an X.509 certificate. note: The content at the address pointed to by any of the parameters that are pointers must not be modified until the certificate is generated and you have the der output. This function does NOT copy the contents to another buffer.

Parameters:

- **cert** Pointer to an initialized DecodedCert object.
- **critical** If 0, the extension will not be marked critical, otherwise it will be marked critical.

- **oid** Dot separated oid as a string. For example “1.2.840.10045.3.1.7”
- **der** The der encoding of the content of the extension.
- **derSz** The size in bytes of the der encoding.

See:

- wc_InitCert
- wc_SetUnknownExtCallback

Return:

- 0 Returned on success.
- Other negative values on failure.

Example

```
int ret = 0;
Cert newCert;
wc_InitCert(&newCert);

// Code to setup subject, public key, issuer, and other things goes here.

ret = wc_SetCustomExtension(&newCert, 1, "1.2.3.4.5",
          (const byte *)"This is a critical extension", 28);
if (ret < 0) {
    // Failed to set the extension.
}

ret = wc_SetCustomExtension(&newCert, 0, "1.2.3.4.6",
          (const byte *)"This is NOT a critical extension", 32)
if (ret < 0) {
    // Failed to set the extension.
}

// Code to sign the certificate and then write it out goes here.
```

```
int wc_SetUnknownExtCallback(
    DecodedCert * cert,
    wc_UnknownExtCallback cb
)
```

This function registers a callback that will be used anytime wolfSSL encounters an unknown X.509 extension in a certificate while parsing a certificate. The prototype of the callback should be:

Parameters:

- **cert** the DecodedCert struct that is to be associated with this callback.
- **cb** function to register as the time callback.

See:

- ParseCert
- wc_SetCustomExtension

Return:

- 0 Returned on success.
- Other negative values on failure.

Example

```
int ret = 0;
// Unknown extension callback prototype
int myUnknownExtCallback(const word16* oid, word32 oidSz, int crit,
                         const unsigned char* der, word32 derSz);

// Register it
ret = wc_SetUnknownExtCallback(cert, myUnknownExtCallback);
if (ret != 0) {
    // failed to set the callback
}

// oid: Array of integers that are the dot separated values in an oid.
// oidSz: Number of values in oid.
// crit: Whether the extension was mark critical.
// der: The der encoding of the content of the extension.
// derSz: The size in bytes of the der encoding.
int myCustomExtCallback(const word16* oid, word32 oidSz, int crit,
                        const unsigned char* der, word32 derSz) {

    // Logic to parse extension goes here.

    // NOTE: by returning zero, we are accepting this extension and
    // informing wolfSSL that it is acceptable. If you find an extension
    // that you do not find acceptable, you should return an error. The
    // standard behavior upon encountering an unknown extension with the
    // critical flag set is to return ASN_CRIT_EXT_E. For the sake of
    // brevity, this example is always accepting every extension; you
    // should use different logic.
    return 0;
}
```

```
int wc_CheckCertSigPubKey(
    const byte * cert,
    word32 certSz,
    void * heap,
    const byte * pubKey,
    word32 pubKeySz,
    int pubKeyOID
)
```

This function verifies the signature in the der form of an X.509 certificate against a public key. The public key is expected to be the full subject public key info in der form.

Parameters:

- **cert** The der encoding of the X.509 certificate.
- **certSz** The size in bytes of cert.
- **heap** A pointer to the heap used for dynamic allocation. Can be NULL.
- **pubKey** The der encoding of the public key.
- **pubKeySz** The size in bytes of pubKey.
- **pubKeyOID** OID identifying the algorithm of the public key. (ie: ECDSAk, DSAk or RSAk)

Return:

- 0 Returned on success.
- Other negative values on failure.

```
int wc_Asn1PrintOptions_Init(
    Asn1PrintOptions * opts
)
```

This function initializes the ASN.1 print options.

Parameters:

- **opts** The ASN.1 options for printing.

See:

- wc_Asn1PrintOptions_Set
- wc_Asn1_PrintAll

Return:

- 0 on success.
- BAD_FUNC_ARG when asn1 is NULL.

Example

```
Asn1PrintOptions opt;

// Initialize ASN.1 print options before use.
wc_Asn1PrintOptions_Init(&opt);
```

```
int wc_Asn1PrintOptions_Set(
    Asn1PrintOptions * opts,
    enum Asn1PrintOpt opt,
    word32 val
)
```

This function sets a print option into an ASN.1 print options object.

Parameters:

- **opts** The ASN.1 options for printing.
- **opt** An option to set value for.
- **val** The value to set.

See:

- wc_Asn1PrintOptions_Init
- wc_Asn1_PrintAll

Return:

- 0 on success.
- BAD_FUNC_ARG when asn1 is NULL.
- BAD_FUNC_ARG when val is out of range for option.

Example

```
Asn1PrintOptions opt;

// Initialize ASN.1 print options before use.
wc_Asn1PrintOptions_Init(&opt);
// Set the number of indents when printing tag name to be 1.
wc_Asn1PrintOptions_Set(&opt, ASN1_PRINT_OPT_INDENT, 1);
```

```
int wc_Asn1_Init(
    Asn1 * asn1
)
```

This function initializes an ASN.1 parsing object.

Parameters:

- **asn1** ASN.1 parse object.

See:

- wc_Asn1_SetFile
- wc_Asn1_PrintAll

Return:

- 0 on success.
- BAD_FUNC_ARG when asn1 is NULL.

Example

```
Asn1 asn1;

// Initialize ASN.1 parse object before use.
wc_Asn1_Init(&asn1);
```

```
int wc_Asn1_SetFile(
    Asn1 * asn1,
    XFILE file
)
```

This function sets the file to use when printing into an ASN.1 parsing object.

Parameters:

- **asn1** The ASN.1 parse object.
- **file** File to print to.

See:

- wc_Asn1_Init
- wc_Asn1_PrintAll

Return:

- 0 on success.
- BAD_FUNC_ARG when asn1 is NULL.
- BAD_FUNC_ARG when file is XBADFILE.

Example

```
Asn1 asn1;

// Initialize ASN.1 parse object before use.
wc_Asn1_Init(&asn1);
// Set standard out to be the file descriptor to write to.
wc_Asn1_SetFile(&asn1, stdout);
```

```
int wc_Asn1_PrintAll(
    Asn1 * asn1,
    Asn1PrintOptions * opts,
    unsigned char * data,
    word32 len
)
```

Print all ASN.1 items.

Parameters:

- **asn1** The ASN.1 parse object.
- **opts** The ASN.1 print options.
- **data** Buffer containing BER/DER data to print.
- **len** Length of data to print in bytes.

See:

- wc_Asn1_Init
- wc_Asn1_SetFile

Return:

- 0 on success.
- BAD_FUNC_ARG when asn1 or opts is NULL.
- ASN_LEN_E when ASN.1 item's length too long.
- ASN_DEPTH_E when end offset invalid.
- ASN_PARSE_E when not all of an ASN.1 item parsed.

```
Asn1PrintOptions opts;
Asn1 asn1;
unsigned char data[] = { Initialize with DER/BER data };
word32 len = sizeof(data);

// Initialize ASN.1 print options before use.
wc_Asn1PrintOptions_Init(&opt);
// Set the number of indents when printing tag name to be 1.
wc_Asn1PrintOptions_Set(&opt, ASN1_PRINT_OPT_INDENT, 1);

// Initialize ASN.1 parse object before use.
wc_Asn1_Init(&asn1);
// Set standard out to be the file descriptor to write to.
wc_Asn1_SetFile(&asn1, stdout);
// Print all ASN.1 items in buffer with the specified print options.
wc_Asn1_PrintAll(&asn1, &opts, data, len);
```

```
int wc_Asn1_SetOidToNameCb(
    Asn1 * asn1,
    Asn1OidToNameCb nameCb
)
```

Sets OID to name callback for ASN.1 parsing.

Parameters:

- **asn1** ASN.1 structure
- **nameCb** Callback function to convert OID to name

See: wc_Asn1_PrintAll

Return:

- 0 on success
- negative on error

Example

```
Asn1 asn1;
int ret = wc_Asn1_SetOidToNameCb(&asn1, myOidToNameCb);
```

```
int wc_GetDecodedCertSubjectRaw(
    const struct DecodedCert * cert,
    const byte ** subjectRaw,
    int * subjectRawSz
)
```

Retrieves the raw DER-encoded subject Name content from a parsed DecodedCert.

Parameters:

- **cert** Pointer to the DecodedCert (must have been parsed).
- **subjectRaw** Output pointer that receives the address of the raw DER subject content.
- **subjectRawSz** Output pointer that receives the size in bytes of the raw subject content.

See:

- wc_InitDecodedCert
- wc_ParseCert
- wc_FreeDecodedCert
- wc_GetDecodedCertSubject
- wolfSSL_UseCertificateAuthority

Return:

- 0 on success.
- BAD_FUNC_ARG if any argument is NULL.
- ASN_PARSE_E if the subject was not populated during parsing.
- NOT_COMPILED_IN if the required build options are not enabled.

The returned pointer and size reference the inner content of the subject Name SEQUENCE (i.e. the bytes after the SEQUENCE tag and length). The pointer aliases the DER buffer supplied to wc_InitDecodedCert(), which the DecodedCert does not own, and must not be freed by the caller. The data remains valid only while that buffer is alive and unmodified.

This function is intended for use with wolfSSL_UseCertificateAuthority(), which expects the subject content without the outer SEQUENCE header.

Requires IGNORE_NAME_CONSTRAINTS to be undefined or WOLFSSL_CERT_EXT to be defined.

Example

```
DecodedCert decoded;
const byte* subject = NULL;
int subjectSz = 0;

wc_InitDecodedCert(&decoded, certDer, certDerSz, NULL);
if (wc_ParseCert(&decoded, CERT_TYPE, NO_VERIFY, NULL) == 0) {
    if (wc_GetDecodedCertSubjectRaw(&decoded, &subject,
            &subjectSz) == 0) {
        // subject and subjectSz now reference the raw DER content
    }
}
wc_FreeDecodedCert(&decoded);
```

B.2 Base Encoding

B.1.2.105 function wc_GetDecodedCertSubjectRaw

B.2.1 Functions

	Name
int	**Base64_Decode**(const byte * in, word32 inLen, byte * out, word32 * outLen)This function decodes the given Base64 encoded input, in, and stores the result in the output buffer out. It also sets the size written to the output buffer in the variable outLen.
int	**Base64_Encode**(const byte * in, word32 inLen, byte * out, word32 * outLen)This function encodes the given input, in, and stores the Base64 encoded result in the output buffer out. It writes the data with the traditional ‘ ’ line endings, instead of escaped %0A line endings. Upon successfully completing, this function also sets outLen to the number of bytes written to the output buffer. If there is enough room in out to store an extra byte, a NULL terminator will be added. This will NOT be included in outLen.

	Name
int	**Base64_EncodeEsc**(const byte * in, word32 inLen, byte * out, word32 * outLen)This function encodes the given input, in, and stores the Base64 encoded result in the output buffer out. It writes the data with %0A escaped line endings instead of ‘ ’ line endings. Upon successfully completing, this function also sets outLen to the number of bytes written to the output buffer.
int	**Base64_Encode_NoNl**(const byte * in, word32 inLen, byte * out, word32 * outLen)This function encodes the given input, in, and stores the Base64 encoded result in the output buffer out. It writes the data with no new lines. Upon successfully completing, this function also sets outLen to the number of bytes written to the output buffer.
int	**Base16_Decode**(const byte * in, word32 inLen, byte * out, word32 * outLen)This function decodes the given Base16 encoded input, in, and stores the result in the output buffer out. It also sets the size written to the output buffer in the variable outLen.
int	**Base16_Encode**(const byte * in, word32 inLen, byte * out, word32 * outLen)Encode input to base16 output. If there is enough room in out to store an extra byte, a NULL terminator will be added and included in outLen.
int	**Base64_Decode_nonCT**(const byte * in, word32 inLen, byte * out, word32 * outLen)This function decodes Base64 encoded input without using constant-time operations. This is faster than the constant-time version but may be vulnerable to timing attacks. Use only when timing attacks are not a concern.
int	**wc_Utf8_DecodeChar**(const byte * in, word32 inLen, word32 * inOutIdx, word32 * cp)This function decodes the UTF-8 encoding of one code point. The encodings that RFC 3629 Section 3 requires a decoder to reject are rejected: overlong forms, the code points reserved for UTF-16 surrogates, code points past the end of the Unicode range, the five and six octet forms that RFC 3629 removed, and sequences whose continuation octets are missing or malformed. Letting any of those decode to a character would allow one octet sequence to impersonate another. The index is only advanced when a code point is decoded, so a caller that wants to continue past a bad sequence chooses for itself how far to skip.

B.2.2 Functions Documentation

```
int Base64_Decode(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

This function decodes the given Base64 encoded input, in, and stores the result in the output buffer out. It also sets the size written to the output buffer in the variable outLen.

Parameters:

- **in** pointer to the input buffer to decode
- **inLen** length of the input buffer to decode
- **out** pointer to the output buffer in which to store the decoded message
- **outLen** pointer to the length of the output buffer. Updated with the bytes written at the end of the function call

See:

- Base64_Encode
- Base16_Decode

Return:

- 0 Returned upon successfully decoding the Base64 encoded input
- BAD_FUNC_ARG Returned if the output buffer is too small to store the decoded input
- ASN_INPUT_E Returned if a character in the input buffer falls outside of the Base64 range ([A-Za-z0-9+/=]) or if there is an invalid line ending in the Base64 encoded input

Example

```
byte encoded[] = { // initialize text to decode };
byte decoded[sizeof(encoded)];
// requires at least (sizeof(encoded) * 3 + 3) / 4 room

int outLen = sizeof(decoded);

if( Base64_Decode(encoded,sizeof(encoded), decoded, &outLen) != 0 ) {
    // error decoding input buffer
}
```

```
int Base64_Encode(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

This function encodes the given input, in, and stores the Base64 encoded result in the output buffer out. It writes the data with the traditional ‘ ’ line endings, instead of escaped %0A line endings. Upon successfully completing, this function also sets outLen to the number of bytes written to the output buffer. If there is enough room in out to store an extra byte, a NULL terminator will be added. This will NOT be included in outLen.

Parameters:

- **in** pointer to the input buffer to encode
- **inLen** length of the input buffer to encode
- **out** pointer to the output buffer in which to store the encoded message
- **outLen** pointer to the length of the output buffer in which to store the encoded message

See:

- Base64_EncodeEsc
- Base64_Decode

Return:

- 0 Returned upon successfully decoding the Base64 encoded input
- BAD_FUNC_ARG Returned if the output buffer is too small to store the encoded input
- BUFFER_E Returned if the output buffer runs out of room while encoding

Example

```
byte plain[] = { // initialize text to encode };
byte encoded[MAX_BUFFER_SIZE];

int outLen = sizeof(encoded);

if( Base64_Encode(plain, sizeof(plain), encoded, &outLen) != 0 ) {
    // error encoding input buffer
}
```

```
int Base64_EncodeEsc(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

This function encodes the given input, in, and stores the Base64 encoded result in the output buffer out. It writes the data with %0A escaped line endings instead of ‘ ’ line endings. Upon successfully completing, this function also sets outLen to the number of bytes written to the output buffer.

Parameters:

- **in** pointer to the input buffer to encode
- **inLen** length of the input buffer to encode

- **out** pointer to the output buffer in which to store the encoded message
- **outLen** pointer to the length of the output buffer in which to store the encoded message

See:

- Base64_Encode
- Base64_Decode

Return:

- 0 Returned upon successfully decoding the Base64 encoded input
- BAD_FUNC_ARG Returned if the output buffer is too small to store the encoded input
- BUFFER_E Returned if the output buffer runs out of room while encoding
- ASN_INPUT_E Returned if there is an error processing the decode on the input message

Example

```
byte plain[] = { // initialize text to encode };
byte encoded[MAX_BUFFER_SIZE];

int outLen = sizeof(encoded);

if( Base64_EncodeEsc(plain, sizeof(plain), encoded, &outLen) != 0 ) {
    // error encoding input buffer
}
```

```
int Base64_Encode_NoNl(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

This function encodes the given input, in, and stores the Base64 encoded result in the output buffer out. It writes the data with no new lines. Upon successfully completing, this function also sets outLen to the number of bytes written to the output buffer.

Parameters:

- **in** pointer to the input buffer to encode
- **inLen** length of the input buffer to encode
- **out** pointer to the output buffer in which to store the encoded message
- **outLen** pointer to the length of the output buffer in which to store the encoded message

See:

- Base64_Encode
- Base64_Decode

Return:

- 0 Returned upon successfully decoding the Base64 encoded input
- BAD_FUNC_ARG Returned if the output buffer is too small to store the encoded input
- BUFFER_E Returned if the output buffer runs out of room while encoding
- ASN_INPUT_E Returned if there is an error processing the decode on the input message

Example

```
byte plain[] = { // initialize text to encode };
byte encoded[MAX_BUFFER_SIZE];
int outLen = sizeof(encoded);
if( Base64_Encode_NoNl(plain, sizeof(plain), encoded, &outLen) != 0 ) {
    // error encoding input buffer
}
```

```
int Base16_Decode(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

This function decodes the given Base16 encoded input, in, and stores the result in the output buffer out. It also sets the size written to the output buffer in the variable outLen.

Parameters:

- **in** pointer to the input buffer to decode
- **inLen** length of the input buffer to decode
- **out** pointer to the output buffer in which to store the decoded message
- **outLen** pointer to the length of the output buffer. Updated with the bytes written at the end of the function call

See:

- Base64_Encode
- Base64_Decode
- Base16_Encode

Return:

- 0 Returned upon successfully decoding the Base16 encoded input
- BAD_FUNC_ARG Returned if the output buffer is too small to store the decoded input or if the input length is not a multiple of two
- ASN_INPUT_E Returned if a character in the input buffer falls outside of the Base16 range ([0-9A-F])

Example

```
byte encoded[] = { // initialize text to decode };
byte decoded[sizeof(encoded)];
int outLen = sizeof(decoded);

if( Base16_Decode(encoded,sizeof(encoded), decoded, &outLen) != 0 ) {
    // error decoding input buffer
}
```

```
int Base16_Encode(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

Encode input to base16 output. If there is enough room in out to store an extra byte, a NULL terminator will be added and included in outLen.

Parameters:

- **in** Pointer to input buffer to be encoded.
- **inLen** Length of input buffer.
- **out** Pointer to output buffer.
- **outLen** Length of output buffer. Is set to len of encoded output.

See:

- Base64_Encode
- Base64_Decode
- Base16_Decode

Return:

- 0 Success
- BAD_FUNC_ARG Returns if in, out, or outLen is null or if outLen is less than 2 times inLen plus 1.

Example

```
byte in[] = { // Contents of something to be encoded };
byte out[NECESSARY_OUTPUT_SIZE];
word32 outSz = sizeof(out);

if(Base16_Encode(in, sizeof(in), out, &outSz) != 0)
{
    // Handle encode error
}
```

```
int Base64_Decode_nonCT(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

This function decodes Base64 encoded input without using constant-time operations. This is faster than the constant-time version but may be vulnerable to timing attacks. Use only when timing attacks are not a concern.

Parameters:

- **in** pointer to the input buffer to decode
- **inLen** length of the input buffer to decode
- **out** pointer to the output buffer to store decoded message
- **outLen** pointer to length of output buffer; updated with bytes written

See:

- Base64_Decode
- Base64_Encode

Return:

- 0 On successfully decoding the Base64 encoded input.
- BAD_FUNC_ARG If the output buffer is too small to store the decoded input.
- ASN_INPUT_E If a character in the input buffer falls outside of the Base64 range or if there is an invalid line ending.
- BUFFER_E If running out of buffer while decoding.

Example

```
byte encoded[] = "SGVsbG8gV29ybGQ="; // "Hello World" in Base64
byte decoded[64];
word32 outLen = sizeof(decoded);

int ret = Base64_Decode_nonCT(encoded, sizeof(encoded)-1, decoded,
                              &outLen);
if (ret != 0) {
    // error decoding input
}
// decoded now contains "Hello World"
```

```
int wc_Utf8_DecodeChar(
    const byte * in,
    word32 inLen,
    word32 * inOutIdx,
    word32 * cp
)
```

This function decodes the UTF-8 encoding of one code point. The encodings that RFC 3629 Section 3 requires a decoder to reject are rejected: overlong forms, the code points reserved for UTF-16 surrogates, code points past the end of the Unicode range, the five and six octet forms that RFC 3629 removed, and sequences whose continuation octets are missing or malformed. Letting any of those decode to a character would allow one octet sequence to impersonate another. The index is only advanced when a code point is decoded, so a caller that wants to continue past a bad sequence chooses for itself how far to skip.

Parameters:

- **in** pointer to the buffer holding UTF-8 encoded text
- **inLen** length of the input buffer in octets
- **inOutIdx** pointer to the index of the first octet to decode; updated to the index of the first octet after the code point decoded
- **cp** pointer to store the decoded code point

Return:

- 0 On successfully decoding one code point.
- BAD_FUNC_ARG If in, inOutIdx or cp is NULL.
- BUFFER_E If no octets remain, or if the sequence needs more continuation octets than the buffer holds.
- ASN_INPUT_E If the octets are not a valid encoding of a code point.

Example

```
const byte text[] = { 0xe2, 0x82, 0xac }; // U+20AC
word32 idx = 0;
word32 cp;

while (idx < sizeof(text)) {
    if (wc_Utf8_DecodeChar(text, sizeof(text), &idx, &cp) != 0) {
        // not valid UTF-8 at idx
        break;
    }
    // cp holds the next code point
}
```

B.3 Compression

B.2.2.8 function wc_Utf8_DecodeChar

B.3.1 Functions

	Name
int	**wc_Compress**(byte * out, word32 outSz, const byte * in, word32 inSz, word32 flags)This function compresses the given input data using Huffman coding and stores the output in out. Note that the output buffer should still be larger than the input buffer because there exists a certain input for which there will be no compression possible, which will still require a lookup table. It is recommended that one allocate srcSz + 0.1% + 12 for the output buffer.
int	**wc_DeCompress**(byte * out, word32 outSz, const byte * in, word32 inSz)This function decompresses the given compressed data using Huffman coding and stores the output in out.
int	**wc_Compress_ex**(byte * out, word32 outSz, const byte * in, word32 inSz, word32 flags, word32 windowBits)This function compresses the given input data using Huffman coding with extended parameters. This is similar to wc_Compress but allows specification of compression flags and window bits for more control over the compression process.
int	**wc_DeCompress_ex**(byte * out, word32 outSz, const byte * in, word32 inSz, int windowBits)This function decompresses the given compressed data using Huffman coding with extended parameters. This is similar to wc_DeCompress but allows specification of window bits for more control over the decompression process.
int	**wc_DeCompressDynamic**(byte ** out, int max, int memoryType, const byte * in, word32 inSz, int windowBits, void * heap)This function decompresses the given compressed data using Huffman coding with dynamic memory allocation. The output buffer is allocated dynamically and the caller is responsible for freeing it.

B.3.2 Functions Documentation

```
int wc_Compress(
    byte * out,
    word32 outSz,
    const byte * in,
    word32 inSz,
    word32 flags
)
```

This function compresses the given input data using Huffman coding and stores the output in out. Note that the output buffer should still be larger than the input buffer because there exists a certain

input for which there will be no compression possible, which will still require a lookup table. It is recommended that one allocate srcSz + 0.1% + 12 for the output buffer.

Parameters:

- **out** pointer to the output buffer in which to store the compressed data
- **outSz** size available in the output buffer for storage
- **in** pointer to the buffer containing the message to compress
- **inSz** size of the input message to compress
- **flags** flags to control how compression operates. Use 0 for normal decompression

See: wc_DeCompress

Return:

- On successfully compressing the input data, returns the number of bytes stored in the output buffer
- COMPRESS_INIT_E Returned if there is an error initializing the stream for compression
- COMPRESS_E Returned if an error occurs during compression

Example

```
byte message[] = { // initialize text to compress };
byte compressed[(sizeof(message) + sizeof(message) * .001 + 12 )];
// Recommends at least srcSz + .1% + 12

if( wc_Compress(compressed, sizeof(compressed), message, sizeof(message),
0) != 0){
    // error compressing data
}
```

```
int wc_DeCompress(
    byte * out,
    word32 outSz,
    const byte * in,
    word32 inSz
)
```

This function decompresses the given compressed data using Huffman coding and stores the output in out.

Parameters:

- **out** pointer to the output buffer in which to store the decompressed data
- **outSz** size available in the output buffer for storage
- **in** pointer to the buffer containing the message to decompress
- **inSz** size of the input message to decompress

See: wc_Compress

Return:

- Success On successfully decompressing the input data, returns the number of bytes stored in the output buffer
- COMPRESS_INIT_E: Returned if there is an error initializing the stream for compression
- COMPRESS_E: Returned if an error occurs during compression

Example

```
byte compressed[] = { // initialize compressed message };
byte decompressed[MAX_MESSAGE_SIZE];

if( wc_DeCompress(decompressed, sizeof(decompressed),
compressed, sizeof(compressed)) != 0 ) {
    // error decompressing data
}
```

```
int wc_Compress_ex(
    byte * out,
    word32 outSz,
    const byte * in,
    word32 inSz,
    word32 flags,
    word32 windowBits
)
```

This function compresses the given input data using Huffman coding with extended parameters. This is similar to wc_Compress but allows specification of compression flags and window bits for more control over the compression process.

Parameters:

- **out** pointer to the output buffer in which to store the compressed data
- **outSz** size available in the output buffer for storage
- **in** pointer to the buffer containing the message to compress
- **inSz** size of the input message to compress
- **flags** flags to control how compression operates
- **windowBits** the base two logarithm of the window size (8..15)

See:

- wc_Compress
- wc_DeCompress_ex

Return:

- On successfully compressing the input data, returns the number of bytes stored in the output buffer
- COMPRESS_INIT_E Returned if there is an error initializing the stream for compression
- COMPRESS_E Returned if an error occurs during compression

Example

```
byte message[] = { // initialize text to compress };
byte compressed[(sizeof(message) + sizeof(message) * .001 + 12)];
word32 flags = 0;
word32 windowBits = 15; // 32KB window

int ret = wc_Compress_ex(compressed, sizeof(compressed), message,
                         sizeof(message), flags, windowBits);
if (ret < 0) {
    // error compressing data
}
```

```
int wc_DeCompress_ex(
    byte * out,
    word32 outSz,
    const byte * in,
    word32 inSz,
    int windowBits
)
```

This function decompresses the given compressed data using Huffman coding with extended parameters. This is similar to wc_DeCompress but allows specification of window bits for more control over the decompression process.

Parameters:

- **out** pointer to the output buffer in which to store the decompressed data
- **outSz** size available in the output buffer for storage
- **in** pointer to the buffer containing the message to decompress
- **inSz** size of the input message to decompress
- **windowBits** the base two logarithm of the window size (8..15)

See:

- wc_DeCompress
- wc_Compress_ex

Return:

- On successfully decompressing the input data, returns the number of bytes stored in the output buffer
- COMPRESS_INIT_E Returned if there is an error initializing the stream for decompression
- COMPRESS_E Returned if an error occurs during decompression

Example

```
byte compressed[] = { // initialize compressed message };
byte decompressed[MAX_MESSAGE_SIZE];
```

```
int windowBits = 15;

int ret = wc_DeCompress_ex(decompressed, sizeof(decompressed),
                           compressed, sizeof(compressed),
                           windowBits);
if (ret < 0) {
    // error decompressing data
}
```

```
int wc_DeCompressDynamic(
    byte ** out,
    int max,
    int memoryType,
    const byte * in,
    word32 inSz,
    int windowBits,
    void * heap
)
```

This function decompresses the given compressed data using Huffman coding with dynamic memory allocation. The output buffer is allocated dynamically and the caller is responsible for freeing it.

Parameters:

- **out** pointer to pointer that will be set to the allocated output buffer
- **max** maximum size to allocate for output buffer
- **memoryType** type of memory to allocate (DYNAMIC_TYPE_TMP_BUFFER)
- **in** pointer to the buffer containing the message to decompress
- **inSz** size of the input message to decompress
- **windowBits** the base two logarithm of the window size (8..15)
- **heap** heap hint for memory allocation (can be NULL)

See:

- wc_DeCompress
- wc_DeCompress_ex

Return:

- On successfully decompressing the input data, returns the number of bytes stored in the output buffer
- COMPRESS_INIT_E Returned if there is an error initializing the stream for decompression
- COMPRESS_E Returned if an error occurs during decompression
- MEMORY_E Returned if memory allocation fails

Example

```
byte compressed[] = { // initialize compressed message };
byte* decompressed = NULL;
int max = 1024 * 1024; // 1MB max
```

```
int ret = wc_DeCompressDynamic(&decompressed, max,
                               DYNAMIC_TYPE_TMP_BUFFER, compressed,
                               sizeof(compressed), 15, NULL);
if (ret < 0) {
    // error decompressing data
}
else {
    // use decompressed data
    XFREE(decompressed, NULL, DYNAMIC_TYPE_TMP_BUFFER);
}
```

B.4 Error Reporting

B.3.2.5 function wc_DeCompressDynamic

B.4.1 Functions

	Name
void	**wc_ErrorString**(int err, char * buff)This function stores the error string for a particular error code in the given buffer.
const char *	**wc_GetErrorString**(int error)This function returns the error string for a particular error code.

B.4.2 Functions Documentation

```
void wc_ErrorString(
    int err,
    char * buff
)
```

This function stores the error string for a particular error code in the given buffer.

Parameters:

- **error** error code for which to get the string
- **buffer** buffer in which to store the error string. Buffer should be at least WOLFSSL_MAX_ERROR_SZ (80 bytes) long

See: wc_GetErrorString

Return: none No returns.

Example

```
char errorMsg[WOLFSSL_MAX_ERROR_SZ];
int err = wc_some_function();
```

```
if( err != 0) { // error occurred
    wc_ErrorString(err, errorMsg);
}
```

```
const char * wc_GetErrorString(
    int error
)
```

This function returns the error string for a particular error code.

Parameters:

- **error** error code for which to get the string

See: wc_ErrorString

Return: string Returns the error string for an error code as a string literal.

Example

```
char * errorMsg;
int err = wc_some_function();

if( err != 0) { // error occurred
    errorMsg = wc_GetErrorString(err);
}
```

B.5 IoT-Safe Module

B.4.2.2 function wc_GetErrorString More...

B.5.1 Functions

	Name
int	**wolfSSL_CTX_iotsafe_enable**(WOLFSSL_CTX * ctx)This function enables the IoT-Safe support on the given context.
int	**wolfSSL_iotsafe_on**(WOLFSSL * ssl, byte privkey_id, byte ecdh_keypair_slot, byte peer_pubkey_slot, byte peer_cert_slot)This function connects the IoT-Safe TLS callbacks to the given SSL session.

	Name
int	**wolfSSL_iotsafe_on_ex**(WOLFSSL * ssl, byte * privkey_id, byte * ecdh_keypair_slot, byte * peer_pubkey_slot, byte * peer_cert_slot, word16 id_size)This function connects the IoT_Safe TLS callbacks to the given SSL session. This is equivalent to wolfSSL_iotsafe_on except that the IDs for the IoT-SAFE slots can be passed by reference, and the length of the ID fields can be specified via the parameter “id_size”.
void	**wolfIoTSafe_SetCSIM_read_cb**(wolfSSL_IOTSafe_CSIM_read_cb rf)Associates a read callback for the AT+CSIM commands. This input function is usually associated to a read event of a UART channel communicating with the modem. The read callback associated is global and changes for all the contexts that use IoT-safe support at the same time. This function must be called from a single thread, during initialization, before any IoT-Safe operation is performed.
void	**wolfIoTSafe_SetCSIM_write_cb**(wolfSSL_IOTSafe_CSIM_write_cb wf)Associates a write callback for the AT+CSIM commands. This output function is usually associated to a write event on a UART channel communicating with the modem. The write callback associated is global and changes for all the contexts that use IoT-safe support at the same time. This function must be called from a single thread, during initialization, before any IoT-Safe operation is performed.
int	**wolfIoTSafe_GetRandom**(unsigned char * out, word32 sz)Generate a random buffer of given size, using the IoT-Safe function GetRandom. This function is automatically used by the wolfCrypt RNG object.
int	**wolfIoTSafe_GetCert**(uint8_t id, unsigned char * output, unsigned long sz)Import a certificate stored in a file on IoT-Safe applet, and store it locally in memory. Works with one-byte file ID field.
int	**wolfIoTSafe_GetCert_ex**(uint8_t * id, uint16_t id_sz, unsigned char * output, unsigned long sz)Import a certificate stored in a file on IoT_Safe applet, and store it locally in memory. Equivalent to wolfIoTSafe_GetCert, except that it can be invoked with a file ID of two or more bytes.
int	**wc_iotsafe_ecc_import_public**(ecc_key * key, byte key_id)Import an ECC 256-bit public key, stored in the IoT-Safe applet, into an ecc_key object.

	Name
int	**wc_iotsafe_ecc_export_public**(ecc_key * key, byte key_id)Export an ECC 256-bit public key, from ecc_key object to a writable public-key slot into the IoT-Safe applet.
int	**wc_iotsafe_ecc_export_public_ex**(ecc_key * key, byte * key_id, word16 id_size)Export an ECC 256-bit public key, from ecc_key object to a writable public-key slot into the IoT-Safe applet. Equivalent to wc_iotsafe_ecc_export_public, except that it can be invoked with a key ID of two or more bytes.
int	**wc_iotsafe_ecc_import_public_ex**(ecc_key * key, byte * key_id, word16 id_size)Export an ECC 256_bit public key, from ecc_key object to a writable public_key slot into the IoT_Safe applet. Equivalent to wc_iotsafe_ecc_import_public, except that it can be invoked with a key ID of two or more bytes.
int	**wc_iotsafe_ecc_export_private**(ecc_key * key, byte key_id)Export an ECC 256-bit key, from ecc_key object to a writable private-key slot into the IoT-Safe applet.
int	**wc_iotsafe_ecc_export_private_ex**(ecc_key * key, byte * key_id, word16 id_size)Export an ECC 256_bit key, from ecc_key object to a writable private_key slot into the IoT_Safe applet. Equivalent to wc_iotsafe_ecc_export_private, except that it can be invoked with a key ID of two or more bytes.
int	**wc_iotsafe_ecc_sign_hash**(byte * in, word32 inlen, byte * out, word32 * outlen, byte key_id)Sign a pre-computed HASH, using a private key previously stored, or pre-provisioned, in the IoT-Safe applet.
int	**wc_iotsafe_ecc_sign_hash_ex**(byte * in, word32 inlen, byte * out, word32 * outlen, byte * key_id, word16 id_size)Sign a pre_computed HASH, using a private key previously stored, or pre_provisioned, in the IoT_Safe applet. Equivalent to wc_iotsafe_ecc_sign_hash, except that it can be invoked with a key ID of two or more bytes.
int	**wc_iotsafe_ecc_verify_hash**(byte * sig, word32 siglen, byte * hash, word32 hashlen, int * res, byte key_id)Verify an ECC signature against a pre-computed HASH, using a public key previously stored, or pre-provisioned, in the IoT-Safe applet. Result is written to res. 1 is valid, 0 is invalid. Note: Do not use the return value to test for valid. Only use res.

	Name
int	**wc_iotsafe_ecc_verify_hash_ex**(byte * sig, word32 siglen, byte * hash, word32 hashlen, int * res, byte * key_id, word16 id_size)Verify an ECC signature against a pre_computed HASH, using a public key previously stored, or pre_provisioned, in the IoT_Safe applet. Result is written to res. 1 is valid, 0 is invalid. Note: Do not use the return value to test for valid. Only use res. Equivalent to wc_iotsafe_ecc_verify_hash, except that it can be invoked with a key ID of two or more bytes.
int	**wc_iotsafe_ecc_gen_k**(byte key_id)Generate an ECC 256_bit keypair and store it in a (writable) slot into the IoT-Safe applet.
int	**wc_iotsafe_ecc_gen_k_ex**(byte * key_id, word16 id_size)Generate an ECC 256_bit keypair and store it in a (writable) slot into the IoT_Safe applet. Equivalent to wc_iotsafe_ecc_gen_k, except that it can be invoked with a key ID of two or more bytes.

B.5.2 Detailed Description

IoT-Safe (IoT-SIM Applet For Secure End-2-End Communication) is a technology that leverage the SIM as robust, scalable and standardized hardware Root of Trust to protect data communication.

IoT-Safe SSL sessions use the SIM as Hardware Security Module, offloading all the crypto public key operations and reducing the attack surface by restricting access to certificate and keys to the SIM.

IoT-Safe support can be enabled on an existing WOLFSSL_CTX context, using wolfSSL_CTX_iotsafe_enable().

Session created within the context can set the parameters for IoT-Safe key and files usage, and enable the public keys callback, with wolfSSL_iotsafe_on().

If compiled in, the module supports IoT-Safe random number generator as source of entropy for wolfCrypt.

B.5.3 Functions Documentation

```
int wolfSSL_CTX_iotsafe_enable(
    WOLFSSL_CTX * ctx
)
```

This function enables the IoT-Safe support on the given context.

Parameters:

- **ctx** pointer to the WOLFSSL_CTX object on which the IoT-safe support must be enabled

See:

- wolfSSL_iotsafe_on
- wolfIoTSafe_SetCSIM_read_cb

- wolfIoTSafe_SetCSIM_write_cb

Return:

- 0 on success
- WC_HW_E on hardware error

This function loads the IoT-Safe applet on first use. In multi-threaded builds both the applet load and every individual APDU transaction are serialized internally on a port-level lock, so concurrent TLS sessions can safely share the same IoT-Safe context, and this function may be called concurrently for several contexts. The CSIM read and write callbacks must still be installed beforehand, from a single thread; see wolfIoTSafe_SetCSIM_read_cb() and wolfIoTSafe_SetCSIM_write_cb().

Example

```
WOLFSSL_CTX *ctx;
ctx = wolfSSL_CTX_new(wolfTLSv1_2_client_method());
if (!ctx)
    return NULL;
wolfSSL_CTX_iotsafe_enable(ctx);
```

```
int wolfSSL_iotsafe_on(
    WOLFSSL * ssl,
    byte privkey_id,
    byte ecdh_keypair_slot,
    byte peer_pubkey_slot,
    byte peer_cert_slot
)
```

This function connects the IoT-Safe TLS callbacks to the given SSL session.

Parameters:

- **ssl** pointer to the WOLFSSL object where the callbacks will be enabled
- **privkey_id** id of the iot-safe applet slot containing the private key for the host
- **ecdh_keypair_slot** id of the iot-safe applet slot to store the ECDH keypair
- **peer_pubkey_slot** id of the iot-safe applet slot to store the other endpoint's public key for ECDH
- **peer_cert_slot** id of the iot-safe applet slot to store the other endpoint's public key for verification

See:

- wolfSSL_iotsafe_on_ex
- wolfSSL_CTX_iotsafe_enable

Return:

- 0 upon success
- NOT_COMPILED_IN if HAVE_PK_CALLBACKS is disabled
- BAD_FUNC_ARG if the ssl pointer is invalid

This should be called to connect a SSL session to IoT-Safe applet when the ID of the slots are one-byte long. If IoT-SAFE slots have an ID of two or more bytes, wolfSSL_iotsafe_on_ex() should be used instead.

Example

```
// Define key ids for IoT-Safe
#define PRIVKEY_ID 0x02
#define ECDH_KEYPAIR_ID 0x03
#define PEER_PUBKEY_ID 0x04
#define PEER_CERT_ID 0x05
// Create new ssl session
WOLFSSL *ssl;
ssl = wolfSSL_new(ctx);
if (!ssl)
    return NULL;
// Enable IoT-Safe and associate key slots
ret = wolfSSL_CTX_iotsafe_enable(ctx);
if (ret == 0) {
    ret = wolfSSL_iotsafe_on(ssl, PRIVKEY_ID, ECDH_KEYPAIR_ID, PEER_PUBKEY_ID,
 ↪  PEER_CERT_ID);
}
```

```
int wolfSSL_iotsafe_on_ex(
    WOLFSSL * ssl,
    byte * privkey_id,
    byte * ecdh_keypair_slot,
    byte * peer_pubkey_slot,
    byte * peer_cert_slot,
    word16 id_size
)
```

This function connects the IoT-Safe TLS callbacks to the given SSL session. This is equivalent to wolfSSL_iotsafe_on except that the IDs for the IoT-SAFE slots can be passed by reference, and the length of the ID fields can be specified via the parameter “id_size”.

Parameters:

- **ssl** pointer to the WOLFSSL object where the callbacks will be enabled
- **privkey_id** pointer to the id of the iot-safe applet slot containing the private key for the host
- **ecdh_keypair_slot** pointer to the id of the iot-safe applet slot to store the ECDH keypair
- **peer_pubkey_slot** pointer to the of id the iot-safe applet slot to store the other endpoint’s public key for ECDH
- **peer_cert_slot** pointer to the id of the iot-safe applet slot to store the other endpoint’s public key for verification
- **id_size** size of each slot ID

See:

- wolfSSL_iotsafe_on
- wolfSSL_CTX_iotsafe_enable

Return:

- 0 upon success
- NOT_COMPILED_IN if HAVE_PK_CALLBACKS is disabled
- BAD_FUNC_ARG if the ssl pointer is invalid

Example

```
// Define key ids for IoT-Safe (16 bit, little endian)
#define PRIVKEY_ID 0x0201
#define ECDH_KEYPAIR_ID 0x0301
#define PEER_PUBKEY_ID 0x0401
#define PEER_CERT_ID 0x0501
#define ID_SIZE (sizeof(word16))

word16 privkey = PRIVKEY_ID,
         ecdh_keypair = ECDH_KEYPAIR_ID,
         peer_pubkey = PEER_PUBKEY_ID,
         peer_cert = PEER_CERT_ID;



// Create new ssl session
WOLFSSL *ssl;
ssl = wolfSSL_new(ctx);
if (!ssl)
    return NULL;
// Enable IoT-Safe and associate key slots
ret = wolfSSL_CTX_iotsafe_enable(ctx);
if (ret == 0) {
    ret = wolfSSL_CTX_iotsafe_on_ex(ssl, &privkey, &ecdh_keypair, &peer_pubkey,
 ↪  &peer_cert, ID_SIZE);
}
```

```
void wolfIoTSafe_SetCSIM_read_cb(
    wolfSSL_IOTSafe_CSIM_read_cb rf
)
```

Associates a read callback for the AT+CSIM commands. This input function is usually associated to a read event of a UART channel communicating with the modem. The read callback associated is global and changes for all the contexts that use IoT-safe support at the same time. This function must be called from a single thread, during initialization, before any IoT-Safe operation is performed.

Parameters:

- **rf** Read callback associated to a UART read event. The callback function takes two arguments (buf, len) and return the number of characters read, up to len. When a newline is encountered, the callback should return the number of characters received so far, including the newline character.

See: wolfIoTSafe_SetCSIM_write_cb

Example

```
// USART read function, defined elsewhere
int usart_read(char *buf, int len);

wolfIoTSafe_SetCSIM_read_cb(usart_read);
```

```
void wolfIoTSafe_SetCSIM_write_cb(
    wolfSSL_IOTSafe_CSIM_write_cb wf
)
```

Associates a write callback for the AT+CSIM commands. This output function is usually associated to a write event on a UART channel communicating with the modem. The write callback associated is global and changes for all the contexts that use IoT-safe support at the same time. This function must be called from a single thread, during initialization, before any IoT-Safe operation is performed.

Parameters:

- **wf** Write callback associated to a UART write event. The callback function takes two arguments (buf, len) and return the number of characters written, up to len.

See: wolfIoTSafe_SetCSIM_read_cb

Example

```
// USART write function, defined elsewhere
int usart_write(const char *buf, int len);
wolfIoTSafe_SetCSIM_write_cb(usart_write);
```

```
int wolfIoTSafe_GetRandom(
    unsigned char * out,
    word32 sz
)
```

Generate a random buffer of given size, using the IoT-Safe function GetRandom. This function is automatically used by the wolfCrypt RNG object.

Parameters:

- **out** the buffer where the random sequence of bytes is stored.
- **sz** the size of the random sequence to generate, in bytes

Return: 0 upon success

```
int wolfIoTSafe_GetCert(
    uint8_t id,
    unsigned char * output,
    unsigned long sz
)
```

Import a certificate stored in a file on IoT-Safe applet, and store it locally in memory. Works with one-byte file ID field.

Parameters:

- **id** The file id in the IoT-Safe applet where the certificate is stored
- **output** the buffer where the certificate will be imported
- **sz** the maximum size available in the buffer output

Return:

- the length of the certificate imported
- < 0 in case of failure

Example

```
#define CRT_CLIENT_FILE_ID 0x03
unsigned char cert_buffer[2048];
// Get the certificate into the buffer
cert_buffer_size = wolfIoTSafe_GetCert(CRT_CLIENT_FILE_ID, cert_buffer, 2048);
if (cert_buffer_size < 1) {
    printf("Bad cli cert\n");
    return -1;
}
printf("Loaded Client certificate from IoT-Safe, size = %lu\n",
 ↪  cert_buffer_size);

// Use the certificate buffer as identity for the TLS client context
if (wolfSSL_CTX_use_certificate_buffer(cli_ctx, cert_buffer,
            cert_buffer_size, SSL_FILETYPE_ASN1) != SSL_SUCCESS) {
    printf("Cannot load client cert\n");
    return -1;
}
printf("Client certificate successfully imported.\n");
```

```
int wolfIoTSafe_GetCert_ex(
    uint8_t * id,
    uint16_t id_sz,
    unsigned char * output,
    unsigned long sz
)
```

Import a certificate stored in a file on IoT-Safe applet, and store it locally in memory. Equivalent to wolfIoTSafe_GetCert, except that it can be invoked with a file ID of two or more bytes.

Parameters:

- **id** Pointer to the file id in the IoT-Safe applet where the certificate is stored
- **id_sz** Size of the file id in bytes
- **output** the buffer where the certificate will be imported
- **sz** the maximum size available in the buffer output

Return:

- the length of the certificate imported
- < 0 in case of failure

Example

```
#define CRT_CLIENT_FILE_ID 0x0302
#define ID_SIZE (sizeof(word16))
unsigned char cert_buffer[2048];
word16 client_file_id = CRT_CLIENT_FILE_ID;



// Get the certificate into the buffer
cert_buffer_size = wolfIoTSafe_GetCert_ex(&client_file_id, ID_SIZE,
 ↪  cert_buffer, 2048);
if (cert_buffer_size < 1) {
    printf("Bad cli cert\n");
    return -1;
}
printf("Loaded Client certificate from IoT-Safe, size = %lu\n",
 ↪  cert_buffer_size);

// Use the certificate buffer as identity for the TLS client context
if (wolfSSL_CTX_use_certificate_buffer(cli_ctx, cert_buffer,
            cert_buffer_size, SSL_FILETYPE_ASN1) != SSL_SUCCESS) {
    printf("Cannot load client cert\n");
    return -1;
}
printf("Client certificate successfully imported.\n");
```

```
int wc_iotsafe_ecc_import_public(
    ecc_key * key,
    byte key_id
)
```

Import an ECC 256-bit public key, stored in the IoT-Safe applet, into an ecc_key object.

Parameters:

- **key** the ecc_key object that will contain the key imported from the IoT-Safe applet
- **id** The key id in the IoT-Safe applet where the public key is stored

See:

- wc_iotsafe_ecc_export_public
- wc_iotsafe_ecc_export_private

Return:

- 0 upon success
- < 0 in case of failure

```
int wc_iotsafe_ecc_export_public(
    ecc_key * key,
    byte key_id
)
```

Export an ECC 256-bit public key, from ecc_key object to a writable public-key slot into the IoT-Safe applet.

Parameters:

- **key** the ecc_key object containing the key to be exported
- **id** The key id in the IoT-Safe applet where the public key will be stored

See:

- wc_iotsafe_ecc_import_public_ex
- wc_iotsafe_ecc_export_private

Return:

- 0 upon success
- < 0 in case of failure

```
int wc_iotsafe_ecc_export_public_ex(
    ecc_key * key,
    byte * key_id,
    word16 id_size
)
```

Export an ECC 256-bit public key, from ecc_key object to a writable public-key slot into the IoT-Safe applet. Equivalent to wc_iotsafe_ecc_export_public, except that it can be invoked with a key ID of two or more bytes.

Parameters:

- **key** the ecc_key object containing the key to be exported
- **key_id** pointer to the key id in the IoT-Safe applet where the public key will be stored

- **id_size** the key id size in bytes

See:

- wc_iotsafe_ecc_export_public
- wc_iotsafe_ecc_import_public_ex
- wc_iotsafe_ecc_export_private_ex

Return:

- 0 upon success
- < 0 in case of failure

Example

```
ecc_key key;
word16 keyId = 0x0302;

wc_ecc_init(&key);
wc_ecc_make_key(&rng, 32, &key);

int ret = wc_iotsafe_ecc_export_public_ex(&key, (byte*)&keyId,
                                          sizeof(keyId));
if (ret != 0) {
    // error exporting public key
}
```

```
int wc_iotsafe_ecc_import_public_ex(
    ecc_key * key,
    byte * key_id,
    word16 id_size
)
```

Export an ECC 256-bit public key, from ecc_key object to a writable public-key slot into the IoT-Safe applet. Equivalent to wc_iotsafe_ecc_import_public, except that it can be invoked with a key ID of two or more bytes.

Parameters:

- **key** the ecc_key object containing the key to be exported
- **id** The pointer to the key id in the IoT-Safe applet where the public key will be stored
- **id_size** The key id size

See:

- wc_iotsafe_ecc_import_public
- wc_iotsafe_ecc_export_private

Return:

- 0 upon success
- < 0 in case of failure

```
int wc_iotsafe_ecc_export_private(
    ecc_key * key,
    byte key_id
)
```

Export an ECC 256-bit key, from ecc_key object to a writable private-key slot into the IoT-Safe applet.

Parameters:

- **key** the ecc_key object containing the key to be exported
- **id** The key id in the IoT-Safe applet where the private key will be stored

See:

- wc_iotsafe_ecc_export_private_ex
- wc_iotsafe_ecc_import_public
- wc_iotsafe_ecc_export_public

Return:

- 0 upon success
- < 0 in case of failure

```
int wc_iotsafe_ecc_export_private_ex(
    ecc_key * key,
    byte * key_id,
    word16 id_size
)
```

Export an ECC 256-bit key, from ecc_key object to a writable private-key slot into the IoT-Safe applet. Equivalent to wc_iotsafe_ecc_export_private, except that it can be invoked with a key ID of two or more bytes.

Parameters:

- **key** the ecc_key object containing the key to be exported
- **id** The pointer to the key id in the IoT-Safe applet where the private key will be stored
- **id_size** The key id size

See:

- wc_iotsafe_ecc_export_private
- wc_iotsafe_ecc_import_public
- wc_iotsafe_ecc_export_public

Return:

- 0 upon success
- < 0 in case of failure

```
int wc_iotsafe_ecc_sign_hash(
    byte * in,
    word32 inlen,
    byte * out,
    word32 * outlen,
    byte key_id
)
```

Sign a pre-computed HASH, using a private key previously stored, or pre-provisioned, in the IoT-Safe applet.

Parameters:

- **in** pointer to the buffer containing the message hash to sign
- **inlen** length of the message hash to sign
- **out** buffer in which to store the generated signature
- **outlen** max length of the output buffer. Will store the bytes
- **id** key id in the IoT-Safe applet for the slot containing the private key to sign the payload written to out upon successfully generating a message signature

See:

- wc_iotsafe_ecc_sign_hash_ex
- wc_iotsafe_ecc_verify_hash
- wc_iotsafe_ecc_gen_k

Return:

- 0 upon success
- < 0 in case of failure

```
int wc_iotsafe_ecc_sign_hash_ex(
    byte * in,
    word32 inlen,
    byte * out,
    word32 * outlen,
    byte * key_id,
    word16 id_size
)
```

Sign a pre-computed HASH, using a private key previously stored, or pre-provisioned, in the IoT-Safe applet. Equivalent to wc_iotsafe_ecc_sign_hash, except that it can be invoked with a key ID of two or more bytes.

Parameters:

- **in** pointer to the buffer containing the message hash to sign
- **inlen** length of the message hash to sign
- **out** buffer in which to store the generated signature
- **outlen** max length of the output buffer. Will store the bytes
- **id** pointer to a key id in the IoT-Safe applet for the slot containing the private key to sign the payload written to out upon successfully generating a message signature
- **id_size** The key id size

See:

- wc_iotsafe_ecc_sign_hash
- wc_iotsafe_ecc_verify_hash
- wc_iotsafe_ecc_gen_k

Return:

- 0 upon success
- < 0 in case of failure

```
int wc_iotsafe_ecc_verify_hash(
    byte * sig,
    word32 siglen,
    byte * hash,
    word32 hashlen,
    int * res,
    byte key_id
)
```

Verify an ECC signature against a pre-computed HASH, using a public key previously stored, or pre-provisioned, in the IoT-Safe applet. Result is written to res. 1 is valid, 0 is invalid. Note: Do not use the return value to test for valid. Only use res.

Parameters:

- **sig** buffer containing the signature to verify
- **hash** The hash (message digest) that was signed
- **hashlen** The length of the hash (octets)
- **res** Result of signature, 1==valid, 0==invalid
- **key_id** The id of the slot where the public ECC key is stored in the IoT-Safe applet

See:

- wc_iotsafe_ecc_verify_hash_ex
- wc_iotsafe_ecc_sign_hash
- wc_iotsafe_ecc_gen_k

Return:

- 0 upon success (even if the signature is not valid)
- < 0 in case of failure.

```
int wc_iotsafe_ecc_verify_hash_ex(
    byte * sig,
    word32 siglen,
    byte * hash,
    word32 hashlen,
    int * res,
    byte * key_id,
    word16 id_size
)
```

Verify an ECC signature against a pre-computed HASH, using a public key previously stored, or pre-provisioned, in the IoT-Safe applet. Result is written to res. 1 is valid, 0 is invalid. Note: Do not use the return value to test for valid. Only use res. Equivalent to wc_iotsafe_ecc_verify_hash, except that it can be invoked with a key ID of two or more bytes.

Parameters:

- **sig** buffer containing the signature to verify
- **hash** The hash (message digest) that was signed
- **hashlen** The length of the hash (octets)
- **res** Result of signature, 1==valid, 0==invalid
- **key_id** The id of the slot where the public ECC key is stored in the IoT-Safe applet
- **id_size** The key id size

See:

- wc_iotsafe_ecc_verify_hash
- wc_iotsafe_ecc_sign_hash
- wc_iotsafe_ecc_gen_k

Return:

- 0 upon success (even if the signature is not valid)
- < 0 in case of failure.

```
int wc_iotsafe_ecc_gen_k(
    byte key_id
)
```

Generate an ECC 256-bit keypair and store it in a (writable) slot into the IoT-Safe applet.

Parameters:

- **key_id** The id of the slot where the ECC key pair is stored in the IoT-Safe applet.

See:

- wc_iotsafe_ecc_gen_k_ex
- wc_iotsafe_ecc_sign_hash
- wc_iotsafe_ecc_verify_hash

Return:

- 0 upon success
- < 0 in case of failure.

```
int wc_iotsafe_ecc_gen_k_ex(
    byte * key_id,
    word16 id_size
)
```

Generate an ECC 256-bit keypair and store it in a (writable) slot into the IoT-Safe applet. Equivalent to wc_iotsafe_ecc_gen_k, except that it can be invoked with a key ID of two or more bytes.

Parameters:

- **key_id** The id of the slot where the ECC key pair is stored in the IoT-Safe applet.
- **id_size** The key id size

See:

- wc_iotsafe_ecc_gen_k
- wc_iotsafe_ecc_sign_hash_ex
- wc_iotsafe_ecc_verify_hash_ex

Return:

- 0 upon success
- < 0 in case of failure.

B.6 Logging

B.5.3.20 function wc_iotsafe_ecc_gen_k_ex

B.6.1 Functions

	Name
int	**wolfSSL_SetLoggingCb**(wolfSSL_Logging_cb log_function)This function registers a logging callback that will be used to handle the wolfSSL log message. By default, if the system supports it fprintf() to stderr is used but by using this function anything can be done by the user.

B.6.2 Functions Documentation

```
int wolfSSL_SetLoggingCb(
    wolfSSL_Logging_cb log_function
)
```

This function registers a logging callback that will be used to handle the wolfSSL log message. By default, if the system supports it fprintf() to stderr is used but by using this function anything can be done by the user.

Parameters:

- **log_function** function to register as a logging callback. Function signature must follow the above prototype.

See:

- wolfSSL_Debugging_ON
- wolfSSL_Debugging_OFF

Return:

- Success If successful this function will return 0.
- BAD_FUNC_ARG is the error that will be returned if a function pointer is not provided.

Example

```
int ret = 0;
// Logging callback prototype
void MyLoggingCallback(const int logLevel, const char* const logMessage);
// Register the custom logging callback with wolfSSL
ret = wolfSSL_SetLoggingCb(MyLoggingCallback);
if (ret != 0) {
    // failed to set logging callback
}
void MyLoggingCallback(const int logLevel, const char* const logMessage)
{
// custom logging function
}
```

B.7 Math API

B.6.2.1 function wolfSSL_SetLoggingCb

B.7.1 Functions

	Name
word32	**CheckRunTimeFastMath**(void)This function checks the runtime fastmath settings for the maximum size of an integer. It is important when a user is using a wolfCrypt library independently, as the FP_SIZE must match for each library in order for math to work correctly. This check is defined as CheckFastMathSettings(), which simply compares CheckRunTimeFastMath and FP_SIZE, returning 0 if there is a mismatch, or 1 if they match.
word32	**CheckRunTimeSettings**(void)This function checks the compile time class settings. It is important when a user is using a wolfCrypt library independently, as the settings must match between libraries for math to work correctly. This check is defined as CheckCtcSettings(), which simply compares CheckRunTimeSettings and CTC_SETTINGS, returning 0 if there is a mismatch, or 1 if they match.

B.7.2 Functions Documentation

```
word32 CheckRunTimeFastMath(
    void
)
```

This function checks the runtime fastmath settings for the maximum size of an integer. It is important when a user is using a wolfCrypt library independently, as the FP_SIZE must match for each library in order for math to work correctly. This check is defined as CheckFastMathSettings(), which simply compares CheckRunTimeFastMath and FP_SIZE, returning 0 if there is a mismatch, or 1 if they match.

Parameters:

- **none** No parameters.

See: CheckRunTimeSettings

Return: FP_SIZE Returns FP_SIZE, corresponding to the max size available for the math library.

Example

```
if (CheckFastMathSettings() != 1) {
return err_sys("Build vs. runtime fastmath FP_MAX_BITS mismatch\n");
```

```
}
// This is converted by the preprocessor to:
// if ( (CheckRunTimeFastMath() == FP_SIZE) != 1) {
// and confirms that the fast math settings match
// the compile time settings



word32 CheckRunTimeSettings(
    void
)
```

This function checks the compile time class settings. It is important when a user is using a wolfCrypt library independently, as the settings must match between libraries for math to work correctly. This check is defined as CheckCtcSettings(), which simply compares CheckRunTimeSettings and CTC_SETTINGS, returning 0 if there is a mismatch, or 1 if they match.

Parameters:

- **none** No Parameters.

See: CheckRunTimeFastMath

Return: settings Returns the runtime CTC_SETTINGS (Compile Time Settings)

Example

```
if (CheckCtcSettings() != 1) {
    return err_sys("Build vs. runtime math mismatch\n");
}
// This is converted by the preprocessor to:
// if ( (CheckCtcSettings() == CTC_SETTINGS) != 1) {
// and will compare whether the compile time class settings
// match the current settings
```

B.8 Random Number Generation

B.7.2.2 function CheckRunTimeSettings More...

B.8.1 Functions

	Name
int	**wc_InitNetRandom**(const char * configFile, wnr_hmac_key hmac_cb, int timeout)Init global Whitewood netRandom context.
int	**wc_FreeNetRandom**(void)Free global Whitewood netRandom context.

	Name
int	**wc_InitRng**(WC_RNG * rng)Gets the seed (from OS) and key cipher for rng. rng_>drbg (deterministic random bit generator) allocated (should be deallocated with wc_FreeRng). This is a blocking operation.
int	**wc_RNG_GenerateBlock**(WC_RNG * rng, byte * output, word32 sz)Copies a sz bytes of pseudorandom data to output. Will reseed rng if needed (blocking).
int	**wc_RNG_GenerateByte**(WC_RNG * rng, byte * b)Calls wc_RNG_GenerateBlock to copy a byte of pseudorandom data to b. Will reseed rng if needed.
int	**wc_FreeRng**(WC_RNG * rng)Should be called when RNG no longer needed in order to securely free drgb. Zeros and XFREEs rng_drbg. The WC_RNG must have come from wc_InitRng*() or be zeroed.
int	**wc_RNG_HealthTest**(int reseed, const byte * seedA, word32 seedASz, const byte * seedB, word32 seedBSz, byte * output, word32 outputSz)Creates and tests functionality of drbg.
int	**wc_GenerateSeed**(OS_Seed * os, byte * output, word32 sz)Generates seed from OS entropy source. Lower-level function used internally by wc_InitRng.
WC_RNG *	**wc_rng_new**(byte * nonce, word32 nonceSz, void * heap)Allocates and initializes new WC_RNG with optional nonce.
int	**wc_rng_new_ex**(WC_RNG ** rng, byte * nonce, word32 nonceSz, void * heap, int devId)Allocates and initializes WC_RNG with extended parameters.
void	**wc_rng_free**(WC_RNG * rng)Frees WC_RNG allocated with wc_rng_new.
int	**wc_InitRng_ex**(WC_RNG * rng, void * heap, int devId)Initializes WC_RNG with extended parameters.
int	**wc_InitRngNonce**(WC_RNG * rng, const byte * nonce, word32 nonceSz)Initializes WC_RNG with nonce.
int	**wc_InitRngNonce_ex**(WC_RNG * rng, const byte * nonce, word32 nonceSz, void * heap, int devId)Initializes WC_RNG with nonce and extended parameters.
int	**wc_SetSeed_Cb**(wc_RngSeed_Cb cb)Sets callback for custom seed generation.
int	**wc_RNG_DRBG_Reseed**(WC_RNG * rng, const byte * seed, word32 seedSz)Reseeds DRBG with new entropy.

	Name
int	**wc_RNG_TestSeed**(const byte * seed, word32 seedSz)Tests seed validity for DRBG.
int	**wc_RNG_HealthTest_ex**(int reseed, const byte * nonce, word32 nonceSz, const byte * seedA, word32 seedASz, const byte * seedB, word32 seedBSz, byte * output, word32 outputSz, void * heap, int devId)RNG health test with extended parameters.
int	**wc_RNG_HealthTest_SHA512**(int reseed, const byte * seedA, word32 seedASz, const byte * seedB, word32 seedBSz, byte * output, word32 outputSz)Runs the SHA_512 Hash_DRBG Known Answer Test (KAT) per SP 800-90A. Instantiates a SHA-512 DRBG with seedA, optionally reseeds with seedB, generates output, and compares against known test vectors. Available when WOLFSSL_DRBG_SHA512 is defined.
int	**wc_RNG_HealthTest_SHA512_ex**(int reseed, const byte * nonce, word32 nonceSz, const byte * persoString, word32 persoStringSz, const byte * seedA, word32 seedASz, const byte * seedB, word32 seedBSz, const byte * additionalA, word32 additionalASz, const byte * additionalB, word32 additionalBSz, byte * output, word32 outputSz, void * heap, int devId)Extended SHA-512 Hash_DRBG health test with nonce, personalization string, and additional input support. Suitable for full ACVP / CAVP test vector validation. Available when WOLFSSL_DRBG_SHA512 is defined.
int	**wc_Sha256Drbg_Disable**(void)Disables the SHA_256 Hash_DRBG at runtime. When disabled, newly initialized WC_RNG instances will not use the SHA_256 DRBG. If the SHA_512 DRBG is enabled (WOLFSSL_DRBG_SHA512), new RNG instances will use SHA-512 instead. Requires HAVE_HASHDRBG.
int	**wc_Sha256Drbg_Enable**(void)Re_enables the SHA_256 Hash_DRBG at runtime after a prior call to wc_Sha256Drbg_Disable(). Requires HAVE_HASHDRBG.
int	**wc_Sha256Drbg_IsDisabled**(void)Returns whether the SHA-256 Hash_DRBG is currently disabled. Requires HAVE_HASHDRBG.

	Name
int	**wc_Sha512Drbg_Disable**(void)Disables the SHA-512 Hash_DRBG at runtime. When disabled, newly initialized WC_RNG instances will not use the SHA-512 DRBG. If the SHA-256 DRBG is still enabled, new RNG instances will fall back to SHA-256. Available when WOLFSSL_DRBG_SHA512 is defined. Requires HAVE_HASHDRBG.
int	**wc_Sha512Drbg_Enable**(void)Re_enables the SHA_512 Hash_DRBG at runtime after a prior call to wc_Sha512Drbg_Disable(). Available when WOLFSSL_DRBG_SHA512 is defined. Requires HAVE_HASHDRBG.
int	**wc_Sha512Drbg_IsDisabled**(void)Returns whether the SHA-512 Hash_DRBG is currently disabled. Available when WOLFSSL_DRBG_SHA512 is defined. Requires HAVE_HASHDRBG.
int	**wc_InitRng_ex2**(WC_RNG * rng, void * heap, int devId, word32 flags)Initialize a WC_RNG with instantiation_time security attributes. Identical to wc_InitRng_ex(), with a flags argument fixing attributes at birth: WC_RNG_INIT_FLAG_LOCK_REQUIRED latches the sticky lock_required policy bit, so there is no reachable state in which the instance serves without its lock policy; WC_RNG_INIT_FLAG_LOCK_INITIALLY constructs into a held lease, to be released with wc_RNG_lock_put(); WC_RNG_INIT_FLAG_USE_FULL_MUTEX layers a blocking wolfSSL_Mutex outermost around the lock latch, for user_mode sharing of one instance among threads (requires WC_RNG_HAVE_LOCK_FULL_MUTEX).
int	**wc_InitRngNonce_ex2**(WC_RNG * rng, const byte * nonce, word32 nonceSz, const byte * perso, word32 persoSz, void * heap, int devId, word32 flags)Initialize a WC_RNG with a caller_supplied nonce and instantiation_time security attributes. The nonce semantics are those of wc_InitRngNonce_ex(); the flags semantics are those of wc_InitRng_ex2().
int	**wc_RNG_GetStatus**(const WC_RNG * rng)Read_only accessor for the RNG health status. Returns the instance's enum wc_RngHealthState value (WC_DRBG_NOT_INIT, WC_DRBG_OK, WC_DRBG_FAILED, WC_DRBG_CONT_FAILED).

	Name
int	**wc_RNG_DRBG_Present**(const WC_RNG * rng)Returns 1 if rng has an instantiated DRBG, else 0. An in_service WC_RNG can lack one: instantiation bypasses the DRBG when the CPU has RDRAND (HAVE_INTEL_RDRAND). DRBG_specific services (commanded reseed, banked next seeds, RBG chains) are unavailable on such instances.
int	**wc_RNG_DRBG_GetReseedCtr**(const WC_RNG * rng, wc_drbg_reseed_ctr_t * reseedCtr)Report the DRBG's current reseed counter – the number of generate operations since the last credited (re)seed, starting at 1.
int	**wc_RNG_DRBG_ScheduleReseed**(WC_RNG * rng)Mark rng due for reseed: the next generate operation reseeds from the module's built-in or registered seed source before producing output. This can only shorten the current seed's remaining lifetime, never extend it.
int	**wc_RNG_DRBG_Reseed_Now**(WC_RNG * rng, const byte * nonce, word32 nonceSz)Immediately reseed rng from the module's built-in or registered seed source, with an optional nonce as additional input. The credited reseed resets the reseed counter.
int	**wc_RNG_DRBG_Reseed_Nonce**(WC_RNG * rng, const byte * seed, word32 seedSz, const byte * nonce, word32 nonceSz)Reseed rng's DRBG with caller-supplied seed material and an optional nonce as additional input. The material is credited as entropy: the reseed counter resets.
int	**wc_RNG_DRBG_Stir**(WC_RNG * rng, const byte * seed, word32 seedSz)Similar to wc_RNG_DRBG_Reseed(), except the caller-supplied material is mixed through the reseed derivation function without being credited as entropy: the reseed counter is not reset, so only the module's own seed source ever extends the instance's seed lifetime.
int	**wc_RNG_DRBG_Stir_Nonce**(WC_RNG * rng, const byte * seed, word32 seedSz, const byte * nonce, word32 nonceSz)The nonce_bearing form of wc_RNG_DRBG_Stir().

	Name
int	**wc_InitRngRBGC**(WC_RNG * child, WC_RNG * parent, word32 flags)Instantiate child as an SP 800_90C RBG_chain member subordinate to parent, drawing its seed material from parent's generate function in place of the module's seed source. Every other aspect of instantiation is wc_InitRng_ex2()'s. The child is tagged with stratum (parent's stratum + 1), sticky for the instance's lifetime even across subsequent source reseeds; its claimable security strength is capped by parent's, and it has no prediction resistance. The caller must hold exclusive access to parent for the duration of the call; the spawn debits parent's reseed counter by one generate.
int	**wc_InitRngNonceRBGC**(WC_RNG * child, WC_RNG * parent, const byte * nonce, word32 nonceSz, word32 flags)The nonce_bearing form of wc_InitRngRBGC(): the nonce is used as additional instantiation input, as in wc_InitRngNonce_ex2().
int	**wc_InitRngRBGC_New**(WC_RNG ** child, WC_RNG * parent, word32 flags)The allocating form of wc_InitRngRBGC(): the child is allocated from parent's heap and returned through child. Release with wc_rng_free().
int	**wc_InitRngNonceRBGC_New**(WC_RNG ** child, WC_RNG * parent, const byte * nonce, word32 nonceSz, word32 flags)The allocating, nonce_bearing form of wc_InitRngRBGC().
int	**wc_RNG_DRBG_ReseedRBGC**(WC_RNG * rng, WC_RNG * root, const byte * nonce, word32 nonceSz)Reseed rng from root's generate output – the SP 800_90C chain reseed – with an optional nonce as additional input. The reseed is credited (the reseed counter resets) and rng acquires root's stratum plus one. The caller must hold exclusive access to both instances.
int	**wc_RNG_DRBG_StirRBGC**(WC_RNG * rng, WC_RNG * root, const byte * nonce, word32 nonceSz)The uncredited form of wc_RNG_DRBG_ReseedRBGC(): material from root is mixed in without resetting rng's reseed counter.
int	**wc_RNG_DRBG_GetRBGCStratum**(const WC_RNG * rng)Report rng's RBG_chain stratum: 0 for a root (never chain_seeded), n for a member seeded from a stratum_(n_1) parent. The stratum is sticky for the instance's lifetime, even across subsequent source reseeds.

	Name
int	**wc_RNG_DRBG_GetNextSeedRBGCStratum**(const WC_RNG * rng)Report the RBG-chain stratum of rng's banked next seed – race-free via the aperture protocol – for provenance-aware consumers.
int	**wc_RNG_DRBG_NextSeedGenerate**(WC_RNG * rng, word32 n)Bank up to n more bytes of next-seed material from the module's seed source, health-testing and publishing the bank when it completes. The fill is incremental and in-boundary; a scheduling daemon may call this without owning the instance – the single-writer fill and the atomic aperture hand-off make it safe alongside a concurrent consumer.
int	**wc_RNG_DRBG_NextSeedGenerate_RBGC**(WC_RNG * rng, WC_RNG * root, word32 n)The chain_sourced form of wc_RNG_DRBG_NextSeedGenerate(): the banked material is drawn from root's generate function, and the bank is tagged with root's stratum plus one for provenance-aware consumption.
int	**wc_RNG_DRBG_NextSeedCurrent**(WC_RNG * rng, WC_ATOMIC_INT_ARG * n)Report the raw next_seed aperture value: a non_negative banked byte count (filling), WC_DRBG_NEXT_SEED_READY, or WC_DRBG_NEXT_SEED_CONSUMING. The snapshot is racy by design; use it for scheduling and diagnostics, not for hand-off decisions.
int	**wc_RNG_DRBG_NextSeedNow**(WC_RNG * rng)Claim a ready next_seed bank and perform a source_free credited reseed with it – safe in atomic context. The bank empties (use_once) and the reseed counter resets. The caller must own the instance.
int	**wc_RNG_DRBG_NextSeedNow_Nonce**(WC_RNG * rng, const byte * nonce, word32 nonceSz)The nonce_bearing form of wc_RNG_DRBG_NextSeedNow(): the nonce is mixed in as uncredited additional input alongside the banked seed.

	Name
int	**wc_RNG_DRBG_NextStirStore**(WC_RNG * rng, const byte * nonce, word32 nonceSz)Bank caller_supplied material (up to WC_RNG_NEXT_STIR_LEN bytes) in the uncredited accumulator beside the banked next seed. Writer_safe without a lease (read_copy_store); if the accumulator is already full, the material is absorbed by xor. Harvested entropy deposited here improves the instance without claiming credit.
int	**wc_RNG_DRBG_NextStirNow**(WC_RNG * rng)Stir the banked uncredited accumulator into the DRBG as an uncredited, source-free mix-in – safe in atomic context; the reseed counter is not reset. The caller must own the instance.
int	**wc_RNG_lock_get**(WC_RNG * rng, WC_RNG_lock_arg_t extra_bits)Acquire rng's exclusive-ownership lock latch, spinning on the CAS until acquired, and or the caller's extra bits into the lock word. On an instance without the lock-required policy the call is a successful no-op unless extra bits are supplied.
int	**wc_RNG_lock_get_conditional**(WC_RNG * rng, WC_RNG_lock_arg_t expected_extra_bits, WC_RNG_lock_arg_t want_extra_bits)The conditional form of wc_RNG_lock_get(): acquire only if the current extra bits equal expected_extra_bits, atomically replacing them with want_extra_bits on success. Non-blocking with respect to the condition: a mismatch fails immediately rather than spinning.
int	**wc_RNG_lock_put**(WC_RNG * rng, WC_RNG_lock_arg_t extra_bits)Release rng's lock latch, clearing the supplied extra bits atomically with the release.
int	**wc_RNG_lock_put_conditional**(WC_RNG * rng, WC_RNG_lock_arg_t expected_extra_bits, WC_RNG_lock_arg_t want_extra_bits)The conditional form of wc_RNG_lock_put(): release only if the current extra bits equal expected_extra_bits, atomically replacing them with want_extra_bits on success.
int	**wc_RNG_lock_read**(WC_RNG * rng, WC_RNG_lock_arg_t * state)Read rng's lock word: the held/required latch bits, the entropy-invalidated bit, and any caller extra bits. The snapshot is racy by design.
int	**wc_RNG_lock_set_extra**(WC_RNG * rng, WC_RNG_lock_arg_t extra_bits)Atomically set (or) the supplied caller extra bits in rng's lock word. The caller should hold the latch.

	Name
int	**wc_RNG_lock_add_extra**(WC_RNG * rng, WC_RNG_lock_arg_t extra_bits)Atomically add the supplied value to the caller extra-bits field of rng's lock word – for counters carried above WC_RNG_LOCK_EXTRA_SHIFT.
int	**wc_RNG_lock_clear_extra**(WC_RNG * rng, WC_RNG_lock_arg_t extra_bits)Atomically clear the supplied caller extra bits in rng's lock word.
int	**wc_RNG_invalidate_entropy**(WC_RNG * rng)Mark rng's seed material untrusted – for VM fork/resume and similar duplication events – opening a new invalidation epoch: banked and pooled pre_event material is purged and wiped first, then the entropy_invalidated bit is latched in the lock word. An invalidated instance refuses service (NEEDS_RECOVERY_E) until recovery_reseeded. Latch or condemn: on any error return the latch is down, and the instance is instead condemned (status DRBG_FAILED). A condemned bank instance is retired and recovered by the entropy daemon; a condemned leaf gets no daemon rescue – its owner sees RNG_FAILURE_E from subsequent operations and recovers it with wc_FreeRng() then wc_InitRng*().
int	**wc_RNG_register_free_hook**(WC_RNG * rng, wc_RNG_free_hook_cb_t free_hook, void * arg)Register a callback fired by wc_FreeRng() at teardown – for external registries (e.g. a kernel_module registry that must reach every live RNG on a VM duplication event) that need to drop their reference when the object dies.
int	**wc_RNG_Pool_Alloc**(WC_RNG * rng, word32 size)Attach a random pool to rng: a buffer of size bytes of pre_generated output, filled by wc_RNG_Pool_Collect() and drained atomic_context_safely by wc_RNG_Pool_Extract(). The pool is released with the instance.
int	**wc_RNG_Pool_Collect**(WC_RNG * rng, word32 n)Generate up to n bytes into rng's pool from rng itself. The collect/extract hand-off is arbitrated by an atomic aperture word, so a single collector is safe alongside concurrent extractors.
int	**wc_RNG_Pool_Collect2**(WC_RNG * rng_dest, WC_RNG * rng_src, word32 n)The two_instance form of wc_RNG_Pool_Collect(): fill rng_dest's pool with output drawn from rng_src – so a service instance's pool can be topped up by a daemon-owned generator.

	Name
int	**wc_RNG_Pool_Extract**(WC_RNG * rng, byte * out, word32 * n)Drain up to *n bytes from rng's pool into out – atomic-context-safe. On success* n reports the bytes actually delivered; on any error return *n is left unmodified.
int	**wc_RNG_Pool_Current**(WC_RNG * rng, word32 * n)Report the pool's current fill in bytes. The snapshot is racy by design.
int	**wc_rng_debug_stats_snap**(struct wc_rng_debug_stats_snapshot * s, const WC_RNG * rng)Snapshot the global RNG debug counters (WC_RNG_DEBUG_STATS) – seeds and reseeds by provenance, generates, pool and bank traffic – into s, for later delta accounting with wc_rng_debug_stats_sum().
int	**wc_rng_debug_stats_restore**(const struct wc_rng_debug_stats_snapshot * s, WC_RNG * rng)Restore the global RNG debug counters from a snapshot – so a test can unwind its own accounting.
int	**wc_rng_debug_stats_sum**(struct wc_rng_debug_stats_snapshot * s, const WC_RNG * rng)Accumulate the current global RNG debug counters into s – combined with a prior wc_rng_debug_stats_snap(), a delta accounting of the interval's RNG activity.
int	**wc_rng_bank_new**(struct wc_rng_bank ** ctx, int n_rngs, word32 flags, int timeout_secs, void * heap, int devId)Allocate and initialize a bank of n_rngs pre_instantiated WC_RNG instances, checked out and back in by consumers (wc_rng_bank_checkout() et al.). The bank is allocated from heap; release with wc_rng_bank_free(). Bank_level flags (WC_RNG_BANK_FLAG_*) fix the bank's posture at initialization: e.g. _CAN_WAIT admits sleeping, _QUIET suppresses seeding-degradation warnings, _NO_CHECKOUT_REFCOUNTING suppresses per-checkout refcount traffic for container-guaranteed lifetimes, _PREDICTION_RESISTANCE imposes a bank-wide fresh-reseed posture on every sleepable lease.
int	**wc_rng_bank_init**(struct wc_rng_bank * ctx, int n_rngs, word32 flags, int timeout_secs, void * heap, int devId)Initialize a caller_provided bank object. Semantics of wc_rng_bank_new(), without the allocation; release with wc_rng_bank_fini().

	Name
int	**wc_rng_bank_init_nonce**(struct wc_rng_bank * ctx, int n_rngs, word32 flags, int timeout_secs, void * heap, int devId, const byte * nonce, word32 nonceSz, const byte * perso, word32 persoSz)The nonce_bearing form of wc_rng_bank_init(): the nonce is used as additional instantiation input for each instance.
int	**wc_rng_bank_first_failover_inst_set**(struct wc_rng_bank * ctx, int first_failover_inst)Designate the first instance of the failover pool: checkouts without a targeted preference rotate through instances at and above first_failover_inst, reserving the lower offsets for targeted (affinity or daemon) use.
int	**wc_rng_bank_set_affinity_handlers**(struct wc_rng_bank * ctx, wc_affinity_lock_fn_t affinity_lock_cb, wc_affinity_get_id_fn_t affinity_get_id_cb, wc_affinity_unlock_fn_t affinity_unlock_cb, void * cb_arg)Install affinity handlers: callbacks that pin the caller to an execution context (e.g. disable preemption or migration), report its id for instance affinity, and unpin. With handlers installed, WC_RNG_BANK_FLAG_PREFER_AFFINITY_INST checkouts prefer the instance matching the caller's affinity id, and WC_RNG_BANK_FLAG_AFFINITY_LOCK holds the pin across the lease.
int	**wc_rng_bank_fini**(struct wc_rng_bank * ctx)Tear down a bank initialized with wc_rng_bank_init(): once the refcount and per-instance lease gates pass, fires any registered free hook and frees every instance. Never waits: a referenced or leased bank is refused with BUSY_E, and the caller quiesces its consumers and retries.
int	**wc_rng_bank_free**(struct wc_rng_bank ** ctx)Tear down and release a bank allocated with wc_rng_bank_new().
int	**wc_rng_bank_default_set**(struct wc_rng_bank * bank)Register bank as the process_default bank, retrievable with wc_rng_bank_default_checkout().
int	**wc_rng_bank_default_checkout**(struct wc_rng_bank ** bank)Take a reference on the process-default bank.
int	**wc_rng_bank_default_checkin**(struct wc_rng_bank ** bank)Release a reference taken with wc_rng_bank_default_checkout().

	Name
int	**wc_rng_bank_default_clear**(struct wc_rng_bank * bank)Unregister the process-default bank.
int	**wc_rng_bank_checkout**(struct wc_rng_bank * bank, struct wc_rng_bank_inst ** rng_inst, int preferred_inst_offset, int timeout_secs, word32 flags)Lease an in_service instance from the bank: either the preferred (or affinity_matched) instance, or – with WC_RNG_BANK_FLAG_CAN_FAIL_OVER_INST – the first available failover instance. On success the caller owns the instance's lock; access the WC_RNG with WC_RNG_BANK_INST_TO_RNG() and return the lease with wc_rng_bank_inst_checkin(). WC_RNG_BANK_FLAG_CONSUME_NEXT_SEED consumes a ready banked next seed in an immediate source_free credited reseed before returning; WC_RNG_BANK_FLAG_PREDICTION_RESISTANCE (per_call, requires _CAN_WAIT) freshly credited-reseeds the lease before the caller's first draw.
int	**wc_rng_bank_checkin**(struct wc_rng_bank * bank, struct wc_rng_bank_inst ** rng_inst)Return a lease through the bank object, validating that rng_inst belongs to bank. Prefer wc_rng_bank_inst_checkin() when only the instance pointer is at hand.
int	**wc_rng_bank_inst_checkin**(struct wc_rng_bank_inst ** rng_inst)Return a lease by instance pointer alone.
int	**wc_rng_bank_get_inst_id**(struct wc_rng_bank_inst * rng_inst)Report the instance's offset within its bank.
int	**wc_rng_bank_next_seed_generate**(struct wc_rng_bank * bank, int inst_offset, word32 n)Bank next_seed material for the instance at inst_offset –wc_RNG_DRBG_NextSeedGenerate() through the bank, without taking the instance lock; the daemon-side serialization word arbitrates against concurrent whole-instance reinitialization.
int	**wc_rng_bank_next_seed_generate_rbgc**(struct wc_rng_bank * bank, int inst_offset, word32 n)The chain_sourced form of wc_rng_bank_next_seed_generate(): the banked material is drawn from root's generate function and tagged with its provenance.

	Name
int	**wc_rng_bank_inst_reinit**(struct wc_rng_bank * bank, struct wc_rng_bank_inst * rng_inst, int timeout_secs, word32 flags)Free and reinstantiate a leased instance in place. The caller must hold the lease; the daemon-side serialization word excludes concurrent lockless banking during the cycle.
int	**wc_rng_bank_recover_inst**(struct wc_rng_bank * bank, int inst_offset, int timeout_secs, word32 flags)Patrol helper: check out the instance at inst_offset with WC_RNG_BANK_FLAG_FOR_RECOVERY, recover it iff it needs recovery, and check it back in. Two recovery arms: an out_of_service instance (wc_RNG_GetStatus() != WC_DRBG_OK) is reinitialized in place; an in_service but entropy_invalidated (quarantined) instance takes one credited reseed, which clears the quarantine while preserving instance identity. A healthy instance is a success no-op, so callers can invoke this unconditionally on state observed locklessly.
int	**wc_rng_bank_spawn**(struct wc_rng_bank * bank, WC_RNG * child_rng, byte * nonce, word32 nonceSz, const byte * perso, word32 persoSz, int preferred_inst_offset, int timeout_secs, word32 flags)Spawn an SP 800_90C chain RNG from a bank instance: check out a parent instance (honoring the usual selection flags), instantiate child_rng as its chain child (wc_InitRngNonceRBGC()), and check the parent back in. The child's lifetime is thereafter decoupled from the parent and its bank; release it with wc_FreeRng() (or wc_rng_free() for the heap form, wc_rng_bank_spawn_new()). The child's RBGC stratum is one plus the parent's stratum at instantiation.
int	**wc_rng_bank_spawn_new**(struct wc_rng_bank * bank, WC_RNG ** child_rng, byte * nonce, word32 nonceSz, const byte * perso, word32 persoSz, int preferred_inst_offset, int timeout_secs, word32 flags)The allocating form of wc_rng_bank_spawn(): the child is allocated from the bank's heap and returned through child_rng; release with wc_rng_free().

	Name
int	**wc_rng_bank_seed**(struct wc_rng_bank * bank, const byte * seed, word32 seedSz, const byte * nonce, word32 nonceSz, int timeout_secs, word32 flags)Reseed every instance with caller-supplied seed material. WC_RNG_BANK_FLAG_STIR mixes the material in without crediting it.
int	**wc_rng_bank_seed_range**(struct wc_rng_bank * bank, int first_inst, int last_inst, const byte * seed, word32 seedSz, const byte * nonce, word32 nonceSz, int timeout_secs, word32 flags)The range form of wc_rng_bank_seed(): seed instances first_inst through last_inst inclusive.
int	**wc_rng_bank_reseed**(struct wc_rng_bank * bank, const byte * nonce, word32 nonceSz, int timeout_secs, word32 flags)Reseed every instance from the module's seed source.
int	**wc_rng_bank_reseed_range**(struct wc_rng_bank * bank, int first_inst, int last_inst, const byte * nonce, word32 nonceSz, int timeout_secs, word32 flags)The range form of wc_rng_bank_reseed().
int	**wc_rng_bank_invalidate_entropy**(struct wc_rng_bank * bank, word32 flags)Set the entropy_invalidated latch on every instance (see wc_RNG_invalidate_entropy()): cached entropy products are discarded, and each instance is forced through a credited reseed before its next generate serves output. Lock_free and constant_time per instance; safe from a state_invalidation event context (VM fork/resume).
int	**wc_rng_bank_daemon_reserve**(struct wc_rng_bank * bank, WC_ATOMIC_UINT_ARG magic)Reserve the bank's daemon slot with a caller-chosen nonzero magic word, admitting exactly one scheduling daemon per bank. The lifecycle is strictly ordered: _reserve, then _register, then _unregister, then _release.
int	**wc_rng_bank_daemon_register**(struct wc_rng_bank * bank, void * daemon, WC_ATOMIC_UINT_ARG magic)Register the daemon object in a slot reserved with the same magic word.
int	**wc_rng_bank_daemon_unregister**(struct wc_rng_bank * bank, void ** daemon, WC_ATOMIC_UINT_ARG magic)Unregister the daemon object, returning it through daemon.

	Name
int	**wc_rng_bank_daemon_release**(struct wc_rng_bank * bank, WC_ATOMIC_UINT_ARG magic)Release the daemon slot claimed with magic, returning it to the free sentinel.
int	**wc_rng_bank_register_free_hook**(struct wc_rng_bank * bank, wc_rng_bank_free_hook_cb_t free_hook, void * arg)Register a callback fired by wc_rng_bank_fini() once its refcount and leak gates pass – i.e. once teardown is committed – for external registries that must drop their reference when the bank dies. One-shot: the hook is cleared before firing. A null free_hook unregisters.
int	**wc_InitRng_BankRef**(struct wc_rng_bank * bank, WC_RNG * rng)Initialize rng as a bank reference: a WC_RNG with no DRBG of its own, whose wc_RNG_GenerateBlock() transparently checks an instance out of bank, generates, and checks it back in. Release with wc_FreeRng().
int	**wc_BankRef_Release**(WC_RNG * rng)Release a bank reference. wc_FreeRng() calls this automatically for bank references; direct use is rarely needed.
int	**wc_rng_new_bankref**(struct wc_rng_bank * bank, WC_RNG ** rng)The allocating form of wc_InitRng_BankRef(): the reference is allocated from the bank's heap; release with wc_rng_free().
int	**wc_rng_bank_debug_stats_snap**(struct wc_rng_debug_stats_snapshot * s, struct wc_rng_bank * bank)Snapshot the RNG debug counters (WC_RNG_DEBUG_STATS) with bank-level context.

B.8.2 Detailed Description

Makes the unguessable numbers used for keys and other secrets. It takes a little true randomness from an entropy source and stretches it into as much output as you ask for.

When that source is wolfEntropy, see Entropy Source - wolfEntropy (MemUse).

B.8.3 Functions Documentation

```
int wc_InitNetRandom(
    const char * configFile,
    wnr_hmac_key hmac_cb,
    int timeout
)
```

Init global Whitewood netRandom context.

Parameters:

- **configFile** Path to configuration file
- **hmac_cb** Optional to create HMAC callback.
- **timeout** A timeout duration.

See: wc_FreeNetRandom

Return:

- 0 Success
- BAD_FUNC_ARG Either configFile is null or timeout is negative.
- RNG_FAILURE_E There was a failure initializing the rng.

Example

```
char* config = "path/to/config/example.conf";
int time = // Some sufficient timeout value;

if (wc_InitNetRandom(config, NULL, time) != 0)
{
    // Some error occurred
}
```

```
int wc_FreeNetRandom(
    void
)
```

Free global Whitewood netRandom context.

Parameters:

- **none** No returns.

See: wc_InitNetRandom

Return:

- 0 Success
- BAD_MUTEX_E Error locking mutex on wnr_mutex

Example

```
int ret = wc_FreeNetRandom();
if(ret != 0)
{
    // Handle the error
}
```

```
int wc_InitRng(
    WC_RNG * rng
)
```

Gets the seed (from OS) and key cipher for rng. rng->drbg (deterministic random bit generator) allocated (should be deallocated with wc_FreeRng). This is a blocking operation.

Parameters:

- **rng** random number generator to be initialized for use with a seed and key cipher

See:

- wc_InitRngCavium
- wc_RNG_GenerateBlock
- wc_RNG_GenerateByte
- wc_FreeRng
- wc_RNG_HealthTest

Return:

- 0 on success.
- MEMORY_E XMALLOC or the fork handler registration failed
- WINCRYPT_E wc_GenerateSeed: failed to acquire context
- CRYPTGEN_E wc_GenerateSeed: failed to get random
- BAD_FUNC_ARG wc_RNG_GenerateBlock input is null or sz exceeds MAX_REQUEST_LEN
- DRBG_CONT_FIPS_E wc_RNG_GenerateBlock: Hash_gen returned DRBG_CONT_FAILURE
- ENTROPY_RT_E or ENTROPY_APT_E wc_InitRng: the SP 800-90B seed health test rejected the entropy gathered to instantiate
- RNG_FAILURE_E wc_RNG_GenerateBlock: Default error. rng's status originally not ok, or set to DRBG_FAILED
- BAD_MUTEX_E the lock that lets threads share this rng could not be created; define WC_RNG_NO_AUTO_LOCK to build without it

One WC_RNG may be shared between threads for generating and reseeding: wc_RNG_GenerateBlock(), wc_RNG_DRBG_Reseed(), wc_RNG_DRBG_Reseed_Nonce() and wc_RNG_DRBG_Reseed_Now() each hold the instance lock. So do the DRBG management calls: wc_RNG_DRBG_Stir(), wc_RNG_DRBG_Stir_Nonce() and wc_RNG_DRBG_ScheduleReseed(), and in –enable_rng_extras builds wc_RNG_DRBG_NextStirNow(), wc_RNG_DRBG_NextSeedNow() and wc_RNG_DRBG_NextSeedNow_Nonce(). The generate path calls their internal cores, which already run under the lock.

wc_RNG_DRBG_ReseedRBGC() and wc_RNG_DRBG_StirRBGC() are the exception: they draw from a second instance, so locking both would put two instance locks in play at once, and two threads reseeding each other from the other could then wait on each other for good. A caller that needs them on a shared instance stops the other threads first.

WC_RNG_NO_AUTO_LOCK (configure –disable-rng-autolock) leaves the lock out; CMSIS-RTOS v1 builds have none, its mutex pool holds ten. Every backend but a crypto callback runs with the lock held; a callback answers first, so it may fall back to the same instance. A seed callback runs with the lock held and must not use the RNG API. wc_InitRng*() and wc_FreeRng() do not lock; initialize only a new or freed WC_RNG, with no other thread using it.

POSIX lets a forked child of a threaded process only exec. Where the build has pthread_atfork(), unnamed POSIX semaphores and the dladdr() pin, fork handlers let the child keep using its WC_RNG: the parent holds every lock across fork() and the child releases them and reseeds, so each fork() waits for

every live instance's generate in flight. The child should use an instance it already has: wc_InitRng() there still waits on a mutex the handlers do not cover. –disable_rng_autofork leaves them out; a user_settings build defines WC_RNG_AUTOFORK to turn them on. The child's reseed uses the configured allocator and seed source, which must therefore work after fork(). wolfCrypt_Init() or the first wc_InitRng() registers them; they can never be unregistered, so the library pins itself against dlclose(). They cover WC_RNG locks only, not clone(), vfork() or _Fork(). A fork() from inside a seed or hash callback works: that thread holds the lock, so the handler leaves it alone and the child frees it as usual. A fork() called from a signal handler can hang the caller, if the signal caught a thread part_way through taking or releasing a lock; POSIX does not allow fork() from a signal handler with handlers like these anyway. Values stay safe either way, with or without the handlers: the child reseeds on its pid change.

Builds without the handlers, macOS among them, leave a forked child only exec(): a child that calls the RNG instead blocks for good if any thread held the instance lock at fork() time. A fork() from a single threaded process is unaffected, since no lock can be held by a thread the child does not have.

Example

```
RNG  rng;
int ret;

#ifdef HAVE_CAVIUM
ret = wc_InitRngCavium(&rng, CAVIUM_DEV_ID);
if (ret != 0){
    printf("RNG Nitrox init for device: %d failed", CAVIUM_DEV_ID);
    return -1;
}
#endif
ret = wc_InitRng(&rng);
if (ret != 0){
    printf("RNG init failed");
    return -1;
}
```

```
int wc_RNG_GenerateBlock(
    WC_RNG * rng,
    byte * output,
    word32 sz
)
```

Copies a sz bytes of pseudorandom data to output. Will reseed rng if needed (blocking).

Parameters:

- **rng** random number generator initialized with wc_InitRng
- **output** buffer to which the block is copied
- **sz** size of output in bytes

See:

- wc_InitRngCavium, wc_InitRng
- wc_RNG_GenerateByte

- wc_FreeRng
- wc_RNG_HealthTest

Return:

- 0 on success
- BAD_FUNC_ARG an input is null or sz exceeds MAX_REQUEST_LEN
- DRBG_CONT_FIPS_E Hash_gen returned DRBG_CONT_FAILURE
- ENTROPY_RT_E or ENTROPY_APT_E the SP 800-90B seed health test rejected the entropy gathered for a reseed
- RNG_FAILURE_E Default error. rng's status originally not ok, or set to DRBG_FAILED
- BAD_MUTEX_E the rng's lock could not be taken

Example

```
RNG  rng;
int  sz = 32;
byte block[sz];

int ret = wc_InitRng(&rng);
if (ret != 0) {
    return -1; //init of rng failed!
}

ret = wc_RNG_GenerateBlock(&rng, block, sz);
if (ret != 0) {
    return -1; //generating block failed!
}
```

```
int wc_RNG_GenerateByte(
    WC_RNG * rng,
    byte * b
)
```

Calls wc_RNG_GenerateBlock to copy a byte of pseudorandom data to b. Will reseed rng if needed.

Parameters:

- **rng** random number generator initialized with wc_InitRng
- **b** one byte buffer to which the block is copied

See:

- wc_InitRngCavium
- wc_InitRng
- wc_RNG_GenerateBlock
- wc_FreeRng
- wc_RNG_HealthTest

Return:

- 0 on success
- BAD_FUNC_ARG an input is null or sz exceeds MAX_REQUEST_LEN
- DRBG_CONT_FIPS_E Hash_gen returned DRBG_CONT_FAILURE
- RNG_FAILURE_E Default error. rng's status originally not ok, or set to DRBG_FAILED
- BAD_MUTEX_E the rng's lock could not be taken

Example

```
RNG  rng;
int  sz = 32;
byte b[1];

int ret = wc_InitRng(&rng);
if (ret != 0) {
    return -1; //init of rng failed!
}

ret = wc_RNG_GenerateByte(&rng, b);
if (ret != 0) {
    return -1; //generating block failed!
}
```

```
int wc_FreeRng(
    WC_RNG * rng
)
```

Should be called when RNG no longer needed in order to securely free drgb. Zeros and XFREEs rng-drbg. The WC_RNG must have come from wc_InitRng*() or be zeroed.

Parameters:

- **rng** random number generator initialized with wc_InitRng

See:

- wc_InitRngCavium
- wc_InitRng
- wc_RNG_GenerateBlock
- wc_RNG_GenerateByte,
- wc_RNG_HealthTest

Return:

- 0 on success
- BAD_FUNC_ARG rng is NULL
- RNG_FAILURE_E Failed to deallocated drbg

Example

```
RNG  rng;
int ret = wc_InitRng(&rng);
if (ret != 0) {
    return -1; //init of rng failed!
}

int ret = wc_FreeRng(&rng);
if (ret != 0) {
    return -1; //free of rng failed!
}
```

```
int wc_RNG_HealthTest(
    int reseed,
    const byte * seedA,
    word32 seedASz,
    const byte * seedB,
    word32 seedBSz,
    byte * output,
    word32 outputSz
)
```

Creates and tests functionality of drbg.

Parameters:

- **int** reseed: if set, will test reseed functionality
- **seedA** seed to instantiate drgb with
- **seedASz** size of seedA in bytes
- **seedB** If reseed set, drbg will be reseeded with seedB
- **seedBSz** size of seedB in bytes
- **output** initialized to random data seeded with seedB if seedrandom is set, and seedA otherwise
- **outputSz** length of output in bytes

See:

- wc_InitRngCavium
- wc_InitRng
- wc_RNG_GenerateBlock
- wc_RNG_GenerateByte
- wc_FreeRng

Return:

- 0 on success
- BAD_FUNC_ARG seedA and output must not be null. If reseed set seedB must not be null
- -1 test failed

Example

```
byte output[SHA256_DIGEST_SIZE * 4];
const byte test1EntropyB[] = ....; // test input for reseed false
const byte test1Output[] = ....;   // testvector: expected output of
                                   // reseed false
ret = wc_RNG_HealthTest(0, test1Entropy, sizeof(test1Entropy), NULL, 0,
                        output, sizeof(output));
if (ret != 0)
    return -1;//healthtest without reseed failed

if (XMEMCMP(test1Output, output, sizeof(output)) != 0)
    return -1; //compare to testvector failed: unexpected output

const byte test2EntropyB[] = ....; // test input for reseed
const byte test2Output[] = ....;   // testvector expected output of reseed
ret = wc_RNG_HealthTest(1, test2EntropyA, sizeof(test2EntropyA),
                        test2EntropyB, sizeof(test2EntropyB),
                        output, sizeof(output));

if (XMEMCMP(test2Output, output, sizeof(output)) != 0)
    return -1; //compare to testvector failed
```

```
int wc_GenerateSeed(
    OS_Seed * os,
    byte * output,
    word32 sz
)
```

Generates seed from OS entropy source. Lower-level function used internally by wc_InitRng.

Parameters:

- **os** Pointer to OS_Seed structure
- **output** Buffer to store seed
- **sz** Size of seed in bytes

See: wc_InitRng

Return:

- 0 On success
- WINCRYPT_E Failed to acquire context (Windows)
- CRYPTGEN_E Failed to generate random (Windows)
- RNG_FAILURE_E Failed to read entropy

Example

```
OS_Seed os;
byte seed[32];
int ret = wc_GenerateSeed(&os, seed, sizeof(seed));
```

```
WC_RNG * wc_rng_new(
    byte * nonce,
    word32 nonceSz,
    void * heap
)
```

Allocates and initializes new WC_RNG with optional nonce.

Parameters:

- **nonce** Nonce buffer (can be NULL)
- **nonceSz** Nonce size
- **heap** Heap hint (can be NULL)

See: wc_rng_free

Return:

- Pointer to WC_RNG on success
- NULL on failure

Example

```
WC_RNG* rng = wc_rng_new(NULL, 0, NULL);
wc_rng_free(rng);
```

```
int wc_rng_new_ex(
    WC_RNG ** rng,
    byte * nonce,
    word32 nonceSz,
    void * heap,
    int devId
)
```

Allocates and initializes WC_RNG with extended parameters.

Parameters:

- **rng** Pointer to store WC_RNG pointer
- **nonce** Nonce buffer (can be NULL)
- **nonceSz** Nonce size
- **heap** Heap hint (can be NULL)
- **devId** Device ID (INVALID_DEVID for software)

See: wc_rng_new

Return:

- 0 On success
- BAD_FUNC_ARG If rng is NULL
- MEMORY_E Memory allocation failed

- BAD_MUTEX_E the lock that lets threads share this rng could not be created

Example

```
WC_RNG* rng;
int ret = wc_rng_new_ex(&rng, NULL, 0, NULL, INVALID_DEVID);
wc_rng_free(rng);
```

```
void wc_rng_free(
    WC_RNG * rng
)
```

Frees WC_RNG allocated with wc_rng_new.

Parameters:

- **rng** WC_RNG to free

See: wc_rng_new

Example

```
WC_RNG* rng = wc_rng_new(NULL, 0, NULL);
wc_rng_free(rng);
```

```
int wc_InitRng_ex(
    WC_RNG * rng,
    void * heap,
    int devId
)
```

Initializes WC_RNG with extended parameters.

Parameters:

- **rng** WC_RNG to initialize
- **heap** Heap hint (can be NULL)
- **devId** Device ID (INVALID_DEVID for software)

See: wc_InitRng

Return:

- 0 On success
- BAD_FUNC_ARG If rng is NULL
- RNG_FAILURE_E Initialization failed
- BAD_MUTEX_E the lock that lets threads share this rng could not be created
- MEMORY_E the fork handlers could not be registered

Example

```
WC_RNG rng;
int ret = wc_InitRng_ex(&rng, NULL, INVALID_DEVID);
wc_FreeRng(&rng);
```

```
int wc_InitRngNonce(
    WC_RNG * rng,
    const byte * nonce,
    word32 nonceSz
)
```

Initializes WC_RNG with nonce.

Parameters:

- **rng** WC_RNG to initialize
- **nonce** Nonce buffer
- **nonceSz** Nonce size

See: wc_InitRng

Return:

- 0 On success
- BAD_FUNC_ARG If rng is NULL
- RNG_FAILURE_E Initialization failed
- BAD_MUTEX_E the lock that lets threads share this rng could not be created
- MEMORY_E the fork handlers could not be registered

Example

```
WC_RNG rng;
byte nonce[16];
int ret = wc_InitRngNonce(&rng, nonce, sizeof(nonce));
wc_FreeRng(&rng);
```

```
int wc_InitRngNonce_ex(
    WC_RNG * rng,
    const byte * nonce,
    word32 nonceSz,
    void * heap,
    int devId
)
```

Initializes WC_RNG with nonce and extended parameters.

Parameters:

- **rng** WC_RNG to initialize
- **nonce** Nonce buffer
- **nonceSz** Nonce size
- **heap** Heap hint (can be NULL)
- **devId** Device ID (INVALID_DEVID for software)

See: wc_InitRngNonce

Return:

- 0 On success
- BAD_FUNC_ARG If rng is NULL
- RNG_FAILURE_E Initialization failed
- BAD_MUTEX_E the lock that lets threads share this rng could not be created
- MEMORY_E the fork handlers could not be registered

Example

```
WC_RNG rng;
byte nonce[16];
int ret = wc_InitRngNonce_ex(&rng, nonce, sizeof(nonce), NULL,
                             INVALID_DEVID);
wc_FreeRng(&rng);
```

```
int wc_SetSeed_Cb(
    wc_RngSeed_Cb cb
)
```

Sets callback for custom seed generation.

Parameters:

- **cb** Seed callback function

See: wc_GenerateSeed

Return:

- 0 On success
- BAD_FUNC_ARG If cb is NULL

Example

```
int my_cb(OS_Seed* os, byte* out, word32 sz) { return 0; }
wc_SetSeed_Cb(my_cb);
```

```
int wc_RNG_DRBG_Reseed(
    WC_RNG * rng,
    const byte * seed,
    word32 seedSz
)
```

Reseeds DRBG with new entropy.

Parameters:

- **rng** WC_RNG to reseed
- **seed** Seed buffer
- **seedSz** Seed size

See: wc_InitRng

Return:

- 0 On success
- BAD_FUNC_ARG If rng or seed is NULL
- RNG_FAILURE_E Reseed failed
- BAD_MUTEX_E the rng's lock could not be taken

Example

```
WC_RNG rng;
byte seed[32];
wc_InitRng(&rng);
int ret = wc_RNG_DRBG_Reseed(&rng, seed, sizeof(seed));
```

```
int wc_RNG_TestSeed(
    const byte * seed,
    word32 seedSz
)
```

Tests seed validity for DRBG.

Parameters:

- **seed** Seed to test
- **seedSz** Seed size

See: wc_InitRng

Return:

- 0 If valid
- BAD_FUNC_ARG If seed is NULL
- ENTROPY_RT_E || ENTROPY_APT_E Validation failed
- ENTROPY_APT_E The adaptive proportion test failed.

Example

```
byte seed[32];
int ret = wc_RNG_TestSeed(seed, sizeof(seed));
```

```
int wc_RNG_HealthTest_ex(
    int reseed,
    const byte * nonce,
    word32 nonceSz,
    const byte * seedA,
    word32 seedASz,
    const byte * seedB,
    word32 seedBSz,
    byte * output,
    word32 outputSz,
    void * heap,
    int devId
)
```

RNG health test with extended parameters.

Parameters:

- **reseed** Non-zero to test reseeding
- **nonce** Nonce buffer (can be NULL)
- **nonceSz** Nonce size
- **seedA** Initial seed
- **seedASz** Initial seed size
- **seedB** Reseed buffer (required if reseed set)
- **seedBSz** Reseed size
- **output** Output buffer
- **outputSz** Output size
- **heap** Heap hint (can be NULL)
- **devId** Device ID (INVALID_DEVID for software)

See: wc_RNG_HealthTest

Return:

- 0 On success
- BAD_FUNC_ARG If required params NULL
- -1 Test failed

Example

```
byte seedA[32], seedB[32], out[64];
int ret = wc_RNG_HealthTest_ex(1, NULL, 0, seedA, 32, seedB, 32,
                               out, 64, NULL, INVALID_DEVID);
```

```
int wc_RNG_HealthTest_SHA512(
    int reseed,
    const byte * seedA,
    word32 seedASz,
    const byte * seedB,
    word32 seedBSz,
    byte * output,
    word32 outputSz
)
```

Runs the SHA-512 Hash_DRBG Known Answer Test (KAT) per SP 800-90A. Instantiates a SHA-512 DRBG with seedA, optionally reseeds with seedB, generates output, and compares against known test vectors. Available when WOLFSSL_DRBG_SHA512 is defined.

Parameters:

- **reseed** Non-zero to test reseeding
- **seedA** Initial entropy seed
- **seedASz** Size of seedA in bytes
- **seedB** Reseed entropy (required if reseed is set)
- **seedBSz** Size of seedB in bytes
- **output** Buffer to receive generated output
- **outputSz** Size of output in bytes

See:

- wc_RNG_HealthTest
- wc_RNG_HealthTest_SHA512_ex

Return:

- 0 On success
- BAD_FUNC_ARG If seedA or output is NULL, or if reseed is set and seedB is NULL
- -1 Test failed

Example

```
byte output[WC_SHA512_DIGEST_SIZE * 4];
const byte seedA[] = { ... };
const byte seedB[] = { ... };

ret = wc_RNG_HealthTest_SHA512(0, seedA, sizeof(seedA), NULL, 0,
                               output, sizeof(output));
if (ret != 0)
    return -1;

ret = wc_RNG_HealthTest_SHA512(1, seedA, sizeof(seedA),
                               seedB, sizeof(seedB),
                               output, sizeof(output));
if (ret != 0)
    return -1;
```

```
int wc_RNG_HealthTest_SHA512_ex(
    int reseed,
    const byte * nonce,
    word32 nonceSz,
    const byte * persoString,
    word32 persoStringSz,
    const byte * seedA,
    word32 seedASz,
    const byte * seedB,
    word32 seedBSz,
    const byte * additionalA,
    word32 additionalASz,
    const byte * additionalB,
    word32 additionalBSz,
    byte * output,
    word32 outputSz,
    void * heap,
    int devId
)
```

Extended SHA-512 Hash_DRBG health test with nonce, personalization string, and additional input support. Suitable for full ACVP / CAVP test vector validation. Available when WOLFSSL_DRBG_SHA512 is defined.

Parameters:

- **reseed** Non-zero to test reseeding
- **nonce** Nonce buffer (can be NULL)
- **nonceSz** Nonce size
- **persoString** Personalization string (can be NULL)
- **persoStringSz** Personalization string size
- **seedA** Initial entropy seed
- **seedASz** Initial seed size
- **seedB** Reseed entropy (required if reseed is set)
- **seedBSz** Reseed size
- **additionalA** Additional input for first generate (can be NULL)
- **additionalASz** Additional input A size
- **additionalB** Additional input for second generate (can be NULL)
- **additionalBSz** Additional input B size
- **output** Output buffer
- **outputSz** Output size
- **heap** Heap hint (can be NULL)
- **devId** Device ID (INVALID_DEVID for software)

See:

- wc_RNG_HealthTest_SHA512
- wc_RNG_HealthTest_ex

Return:

- 0 On success
- BAD_FUNC_ARG If required params are NULL

- -1 Test failed

Example

```
byte output[WC_SHA512_DIGEST_SIZE * 4];
const byte seedA[] = { ... };
const byte nonce[] = { ... };

int ret = wc_RNG_HealthTest_SHA512_ex(0, nonce, sizeof(nonce),
                                      NULL, 0,
                                      seedA, sizeof(seedA),
                                      NULL, 0,
                                      NULL, 0, NULL, 0,
                                      output, sizeof(output),
                                      NULL, INVALID_DEVID);
```

```
int wc_Sha256Drbg_Disable(
    void
)
```

Disables the SHA-256 Hash_DRBG at runtime. When disabled, newly initialized WC_RNG instances will not use the SHA-256 DRBG. If the SHA-512 DRBG is enabled (WOLFSSL_DRBG_SHA512), new RNG instances will use SHA-512 instead. Requires HAVE_HASHDRBG.

See:

- wc_Sha256Drbg_Enable
- wc_Sha256Drbg_IsDisabled
- wc_Sha512Drbg_Disable

Return: 0 On success

Example

```
wc_Sha256Drbg_Disable();
// New WC_RNG instances will now use SHA-512 DRBG if available
WC_RNG rng;
wc_InitRng(&rng);
```

```
int wc_Sha256Drbg_Enable(
    void
)
```

Re-enables the SHA-256 Hash_DRBG at runtime after a prior call to wc_Sha256Drbg_Disable(). Requires HAVE_HASHDRBG.

See:

- wc_Sha256Drbg_Disable

- wc_Sha256Drbg_IsDisabled

Return: 0 On success

Example

```
wc_Sha256Drbg_Disable();
// ... use SHA-512 DRBG only ...
wc_Sha256Drbg_Enable();
// New WC_RNG instances can use SHA-256 DRBG again
```

```
int wc_Sha256Drbg_IsDisabled(
    void
)
```

Returns whether the SHA-256 Hash_DRBG is currently disabled. Requires HAVE_HASHDRBG.

See:

- wc_Sha256Drbg_Disable
- wc_Sha256Drbg_Enable

Return:

- 1 SHA-256 DRBG is disabled
- 0 SHA-256 DRBG is enabled (not disabled)

Example

```
if (wc_Sha256Drbg_IsDisabled()) {
    printf("SHA-256 DRBG is off\n");
}
```

```
int wc_Sha512Drbg_Disable(
    void
)
```

Disables the SHA-512 Hash_DRBG at runtime. When disabled, newly initialized WC_RNG instances will not use the SHA-512 DRBG. If the SHA-256 DRBG is still enabled, new RNG instances will fall back to SHA-256. Available when WOLFSSL_DRBG_SHA512 is defined. Requires HAVE_HASHDRBG.

See:

- wc_Sha512Drbg_Enable
- wc_Sha512Drbg_IsDisabled
- wc_Sha256Drbg_Disable

Return: 0 On success

Example

```
wc_Sha512Drbg_Disable();
// New WC_RNG instances will now use SHA-256 DRBG
WC_RNG rng;
wc_InitRng(&rng);
```

```
int wc_Sha512Drbg_Enable(
    void
)
```

Re-enables the SHA-512 Hash_DRBG at runtime after a prior call to wc_Sha512Drbg_Disable(). Available when WOLFSSL_DRBG_SHA512 is defined. Requires HAVE_HASHDRBG.

See:

- wc_Sha512Drbg_Disable
- wc_Sha512Drbg_IsDisabled

Return: 0 On success

Example

```
wc_Sha512Drbg_Disable();
// ... use SHA-256 DRBG only ...
wc_Sha512Drbg_Enable();
// New WC_RNG instances can use SHA-512 DRBG again
```

```
int wc_Sha512Drbg_IsDisabled(
    void
)
```

Returns whether the SHA-512 Hash_DRBG is currently disabled. Available when WOLFSSL_DRBG_SHA512 is defined. Requires HAVE_HASHDRBG.

See:

- wc_Sha512Drbg_Disable
- wc_Sha512Drbg_Enable

Return:

- 1 SHA-512 DRBG is disabled
- 0 SHA-512 DRBG is enabled (not disabled)

Example

```
if (wc_Sha512Drbg_IsDisabled()) {
    printf("SHA-512 DRBG is off\n");
}
```

```
int wc_InitRng_ex2(
    WC_RNG * rng,
    void * heap,
    int devId,
    word32 flags
)
```

Initialize a WC_RNG with instantiation-time security attributes. Identical to wc_InitRng_ex(), with a flags argument fixing attributes at birth: WC_RNG_INIT_FLAG_LOCK_REQUIRED latches the sticky lock_required policy bit, so there is no reachable state in which the instance serves without its lock policy; WC_RNG_INIT_FLAG_LOCK_INITIALLY constructs into a held lease, to be released with wc_RNG_lock_put(); WC_RNG_INIT_FLAG_USE_FULL_MUTEX layers a blocking wolfSSL_Mutex outermost around the lock latch, for user_mode sharing of one instance among threads (requires WC_RNG_HAVE_LOCK_FULL_MUTEX).

Parameters:

- **rng** The RNG object to initialize.
- **heap** Heap hint for dynamic allocation.
- **devId** Device id, or INVALID_DEVID.
- **flags** Bitwise-or of WC_RNG_INIT_FLAG_* attributes.

See:

- wc_InitRng_ex
- wc_InitRngNonce_ex2
- wc_RNG_lock_get
- wc_RNG_lock_put

Return:

- 0 Success
- BAD_FUNC_ARG rng is null, or the flags contradict each other.
- NOT_COMPILED_IN A requested flag is not compiled in.

WC_RNG_INIT_FLAG_USE_AUTO_LOCK and WC_RNG_INIT_FLAG_NO_AUTO_LOCK decide, for this instance alone, whether the automatic per-call lock is created, whatever –enable-rng-autolock settled for the build. Use the first when the instance will be shared between threads: a build without that lock refuses with NOT_COMPILED_IN rather than returning an instance the caller would wrongly believe is serialized. Use the second to drop the per-call cost on an instance one thread owns; such an instance behaves as it does with the lock left out of the build, so sharing it between threads is a data race and, where the fork handlers exist, it is not fork covered. Setting both, or the first alongside WC_RNG_INIT_FLAG_USE_FULL_MUTEX, is BAD_FUNC_ARG in every build.

Naming neither leaves it to the build: on by default, or off where –disable-rng-autolock-default (WC_RNG_AUTO_LOCK_DEFAULT_OFF) built the lock in without switching it on. In such a build only the two flag forms, wc_InitRng_ex2() and wc_InitRngNonce_ex2(), can ask for the lock; wc_InitRng(),

wc_InitRng_ex(), wc_rng_new() and wc_rng_new_ex() take no flags, so their instances go unlocked and cannot opt in.

Example

```
WC_RNG rng;
if (wc_InitRng_ex2(&rng, NULL, INVALID_DEVID,
                   WC_RNG_INIT_FLAG_LOCK_REQUIRED |
                   WC_RNG_INIT_FLAG_LOCK_INITIALLY) != 0) {
    // error handling
}
// caller holds the lease from birth
```

```
int wc_InitRngNonce_ex2(
    WC_RNG * rng,
    const byte * nonce,
    word32 nonceSz,
    const byte * perso,
    word32 persoSz,
    void * heap,
    int devId,
    word32 flags
)
```

Initialize a WC_RNG with a caller-supplied nonce and instantiation-time security attributes. The nonce semantics are those of wc_InitRngNonce_ex(); the flags semantics are those of wc_InitRng_ex2().

Parameters:

- **rng** The RNG object to initialize.
- **nonce** Optional nonce used as additional instantiation input.
- **nonceSz** Length of nonce in bytes.
- **heap** Heap hint for dynamic allocation.
- **devId** Device id, or INVALID_DEVID.
- **perso** Optional personalization string (may be null).
- **persoSz** Length of perso in bytes.
- **flags** Bitwise-or of WC_RNG_INIT_FLAG_* attributes.

See:

- wc_InitRng_ex2
- wc_InitRngNonce_ex

Return:

- 0 Success
- BAD_FUNC_ARG rng is null.
- NOT_COMPILED_IN A requested flag is not compiled in.

```
int wc_RNG_GetStatus(
    const WC_RNG * rng
)
```

Read-only accessor for the RNG health status. Returns the instance's enum wc_RngHealthState value (WC_DRBG_NOT_INIT, WC_DRBG_OK, WC_DRBG_FAILED, WC_DRBG_CONT_FAILED).

Parameters:

- **rng** The RNG object to interrogate.

See:

- wc_RNG_DRBG_Present
- wc_RNG_DRBG_GetReseedCtr

Return:

- WC_DRBG_OK The instance is in service.
- BAD_FUNC_ARG rng is null.

Example

```
if (wc_RNG_GetStatus(&rng) != WC_DRBG_OK) {
    // instance is not serviceable
}
```

```
int wc_RNG_DRBG_Present(
    const WC_RNG * rng
)
```

Returns 1 if rng has an instantiated DRBG, else 0. An in-service WC_RNG can lack one: instantiation bypasses the DRBG when the CPU has RDRAND (HAVE_INTEL_RDRAND). DRBG-specific services (commanded reseed, banked next seeds, RBG chains) are unavailable on such instances.

Parameters:

- **rng** The RNG object to interrogate.

See:

- wc_RNG_GetStatus
- wc_RNG_DRBG_ScheduleReseed

Return:

- 1 rng has a live DRBG.
- 0 rng is null or has no DRBG.

```
int wc_RNG_DRBG_GetReseedCtr(
    const WC_RNG * rng,
    wc_drbg_reseed_ctr_t * reseedCtr
)
```

Report the DRBG's current reseed counter – the number of generate operations since the last credited (re)seed, starting at 1.

Parameters:

- **rng** The RNG object to interrogate.
- **reseedCtr** Receives the counter.

See:

- wc_RNG_DRBG_Present
- wc_RNG_DRBG_ScheduleReseed

Return:

- 0 Success
- BAD_FUNC_ARG rng or reseedCtr is null.
- WRONG_TYPE_OBJECT_E rng has no DRBG (RDRAND et al.).

```
int wc_RNG_DRBG_ScheduleReseed(
    WC_RNG * rng
)
```

Mark rng due for reseed: the next generate operation reseeds from the module's built-in or registered seed source before producing output. This can only shorten the current seed's remaining lifetime, never extend it.

Parameters:

- **rng** The RNG object to schedule.

See:

- wc_RNG_DRBG_Reseed_Now
- wc_RNG_DRBG_GetReseedCtr

Return:

- 0 Success
- BAD_FUNC_ARG rng is null.
- WRONG_TYPE_OBJECT_E rng has no DRBG (RDRAND et al.) – a commanded reseed that cannot happen is not a success.

```
int wc_RNG_DRBG_Reseed_Now(
    WC_RNG * rng,
    const byte * nonce,
    word32 nonceSz
)
```

Immediately reseed rng from the module's built-in or registered seed source, with an optional nonce as additional input. The credited reseed resets the reseed counter.

Parameters:

- **rng** The RNG object to reseed.
- **nonce** Optional additional input.
- **nonceSz** Length of nonce in bytes.

See:

- wc_RNG_DRBG_ScheduleReseed
- wc_RNG_DRBG_Reseed_Nonce

Return:

- 0 Success
- BAD_FUNC_ARG rng is null, or nonce is null with nonceSz nonzero.
- WRONG_TYPE_OBJECT_E rng has no DRBG (RDRAND et al.).
- DRBG_CONT_FIPS_E The continuous test failed; the DRBG is out of service.
- RNG_FAILURE_E The DRBG is out of service or reseeding failed.
- ENTROPY_RT_E or ENTROPY_APT_E The SP 800-90B seed health test rejected the gathered seed; the reseed did not happen and the instance remains usable until its reseed interval is reached.

```
int wc_RNG_DRBG_Reseed_Nonce(
    WC_RNG * rng,
    const byte * seed,
    word32 seedSz,
    const byte * nonce,
    word32 nonceSz
)
```

Reseed rng's DRBG with caller-supplied seed material and an optional nonce as additional input. The material is credited as entropy: the reseed counter resets.

Parameters:

- **rng** The RNG object to reseed.
- **seed** Seed material.
- **seedSz** Length of seed in bytes.
- **nonce** Optional additional input.
- **nonceSz** Length of nonce in bytes.

See:

- wc_RNG_DRBG_Reseed
- wc_RNG_DRBG_Stir_Nonce

Return:

- 0 Success
- RNG_FAILURE_E rng is condemned (status DRBG_FAILED): a condemned instance does not accept a credited reseed; recover with wc_FreeRng() then wc_InitRng*().
- BAD_FUNC_ARG rng or seed is null.
- WRONG_TYPE_OBJECT_E rng has no DRBG (RDRAND et al.).

```
int wc_RNG_DRBG_Stir(
    WC_RNG * rng,
    const byte * seed,
    word32 seedSz
)
```

Similar to wc_RNG_DRBG_Reseed(), except the caller-supplied material is mixed through the reseed derivation function without being credited as entropy: the reseed counter is not reset, so only the module's own seed source ever extends the instance's seed lifetime.

Parameters:

- **rng** The RNG object to stir.
- **seed** Material to mix in.
- **seedSz** Length of seed in bytes.

See:

- wc_RNG_DRBG_Reseed
- wc_RNG_DRBG_Stir_Nonce

Return:

- 0 Success
- BAD_FUNC_ARG rng or seed is null.
- WRONG_TYPE_OBJECT_E rng has no DRBG (RDRAND et al.).

```
int wc_RNG_DRBG_Stir_Nonce(
    WC_RNG * rng,
    const byte * seed,
    word32 seedSz,
    const byte * nonce,
    word32 nonceSz
)
```

The nonce-bearing form of wc_RNG_DRBG_Stir().

Parameters:

- **rng** The RNG object to stir.
- **seed** Material to mix in.
- **seedSz** Length of seed in bytes.
- **nonce** Optional additional input.
- **nonceSz** Length of nonce in bytes.

See:

- wc_RNG_DRBG_Stir
- wc_RNG_DRBG_Reseed_Nonce

Return:

- 0 Success
- BAD_FUNC_ARG rng or seed is null.
- WRONG_TYPE_OBJECT_E rng has no DRBG (RDRAND et al.).

```
int wc_InitRngRBGC(
    WC_RNG * child,
    WC_RNG * parent,
    word32 flags
)
```

Instantiate child as an SP 800-90C RBG-chain member subordinate to parent, drawing its seed material from parent's generate function in place of the module's seed source. Every other aspect of instantiation is wc_InitRng_ex2()'s. The child is tagged with stratum (parent's stratum + 1), sticky for the instance's lifetime even across subsequent source reseeds; its claimable security strength is capped by parent's, and it has no prediction resistance. The caller must hold exclusive access to parent for the duration of the call; the spawn debits parent's reseed counter by one generate.

Parameters:

- **child** The caller-provided WC_RNG to instantiate (uninitialized).
- **parent** The chain parent to draw seed material from.
- **flags** Bitwise-or of WC_RNG_INIT_FLAG_* attributes for the child.

See:

- wc_InitRngNonceRBGC
- wc_InitRngRBGC_New
- wc_RNG_DRBG_ReseedRBGC
- wc_RNG_DRBG_GetRBGCStratum

Return:

- 0 Success
- BAD_FUNC_ARG child or parent is null, or child equals parent.
- SEQ_OVERFLOW_E parent's stratum is at the representable maximum.

Example

```
WC_RNG root, child;
wc_InitRng(&root);
if (wc_InitRngRBGC(&child, &root, WC_RNG_INIT_FLAG_NONE) == 0) {
    // child serves independently; release with wc_FreeRng(&child)
}
```

```
int wc_InitRngNonceRBGC(
    WC_RNG * child,
    WC_RNG * parent,
    const byte * nonce,
    word32 nonceSz,
    word32 flags
)
```

The nonce-bearing form of wc_InitRngRBGC(): the nonce is used as additional instantiation input, as in wc_InitRngNonce_ex2().

Parameters:

- **child** The caller-provided WC_RNG to instantiate (uninitialized).
- **parent** The chain parent to draw seed material from.
- **nonce** Additional instantiation input.
- **nonceSz** Length of nonce in bytes.
- **flags** Bitwise-or of WC_RNG_INIT_FLAG_* attributes for the child.

See:

- wc_InitRngRBGC
- wc_InitRngNonceRBGC_New

Return:

- 0 Success
- BAD_FUNC_ARG child or parent is null, child equals parent, or nonce is null with nonceSz nonzero.
- SEQ_OVERFLOW_E parent's stratum is at the representable maximum.

```
int wc_InitRngRBGC_New(
    WC_RNG ** child,
    WC_RNG * parent,
    word32 flags
)
```

The allocating form of wc_InitRngRBGC(): the child is allocated from parent's heap and returned through child. Release with wc_rng_free().

Parameters:

- **child** Receives the allocated, instantiated WC_RNG.
- **parent** The chain parent to draw seed material from.
- **flags** Bitwise-or of WC_RNG_INIT_FLAG_* attributes for the child.

See:

- wc_InitRngRBGC
- wc_InitRngNonceRBGC_New

Return:

- 0 Success
- BAD_FUNC_ARG child or parent is null.
- MEMORY_E Allocation failed.
- SEQ_OVERFLOW_E parent's stratum is at the representable maximum.

```
int wc_InitRngNonceRBGC_New(
    WC_RNG ** child,
    WC_RNG * parent,
    const byte * nonce,
    word32 nonceSz,
    word32 flags
)
```

The allocating, nonce-bearing form of wc_InitRngRBGC().

Parameters:

- **child** Receives the allocated, instantiated WC_RNG.
- **parent** The chain parent to draw seed material from.
- **nonce** Additional instantiation input.
- **nonceSz** Length of nonce in bytes.
- **flags** Bitwise-or of WC_RNG_INIT_FLAG_* attributes for the child.

See:

- wc_InitRngRBGC_New
- wc_InitRngNonceRBGC

Return:

- 0 Success
- BAD_FUNC_ARG child or parent is null, or nonce is null with nonceSz nonzero.
- MEMORY_E Allocation failed.
- SEQ_OVERFLOW_E parent's stratum is at the representable maximum.

```
int wc_RNG_DRBG_ReseedRBGC(
    WC_RNG * rng,
    WC_RNG * root,
    const byte * nonce,
    word32 nonceSz
)
```

Reseed rng from root's generate output – the SP 800-90C chain reseed – with an optional nonce as additional input. The reseed is credited (the reseed counter resets) and rng acquires root's stratum plus one. The caller must hold exclusive access to both instances.

Parameters:

- **rng** The chain member to reseed.
- **root** The chain parent to draw seed material from.
- **nonce** Optional additional input.
- **nonceSz** Length of nonce in bytes.

See:

- wc_InitRngRBGC
- wc_RNG_DRBG_StirRBGC
- wc_RNG_DRBG_Reseed_Now

Return:

- 0 Success
- BAD_FUNC_ARG rng or root is null, rng equals root, or the no-downgrade rule refuses root as a chain parent (see
- WRONG_TYPE_OBJECT_E rng has no DRBG (RDRAND et al.).
- SEQ_OVERFLOW_E root's stratum is at the representable maximum.

Credited chain reseeds obey a no-downgrade rule: a primary-seeded (stratum-0) root is always accepted, and a chained (stratum > 0) root is accepted only when its stratum is strictly less than rng's – the acquired stratum never increases, so reseed cycles are impossible by construction, consistent with SP 800-90C 7.1.2.2. Lateral (equal-stratum) and downgrading reseeds are refused with BAD_FUNC_ARG. Building WC_RNG_NO_RBGC_RESEED restricts credited chain reseeds to primary-seeded roots. Uncredited chain stirs (wc_RNG_DRBG_StirRBGC()) are exempt from all of this: they are stirs, claim nothing, and leave rng's stratum untouched.

).

```
int wc_RNG_DRBG_StirRBGC(
    WC_RNG * rng,
    WC_RNG * root,
    const byte * nonce,
    word32 nonceSz
)
```

The uncredited form of wc_RNG_DRBG_ReseedRBGC(): material from root is mixed in without resetting rng's reseed counter.

Parameters:

- **rng** The chain member to stir.
- **root** The chain parent to draw material from.
- **nonce** Optional additional input.
- **nonceSz** Length of nonce in bytes.

See:

- wc_RNG_DRBG_ReseedRBGC
- wc_RNG_DRBG_Stir

Return:

- 0 Success
- BAD_FUNC_ARG rng or root is null, or rng equals root.
- WRONG_TYPE_OBJECT_E rng has no DRBG (RDRAND et al.).

Unrestricted by the credited no-downgrade rule: any source stratum is accepted, and rng's reseed counter, stratum, and entropy-invalidated state are all left untouched – an uncredited chain reseed is a stir, and a stir must never masquerade as recovery or promotion.

```
int wc_RNG_DRBG_GetRBGCStratum(
    const WC_RNG * rng
)
```

Report rng's RBG-chain stratum: 0 for a root (never chain-seeded), n for a member seeded from a stratum-(n-1) parent. The stratum is sticky for the instance's lifetime, even across subsequent source reseeds.

Parameters:

- **rng** The RNG object to interrogate.

See:

- wc_InitRngRBGC
- wc_RNG_DRBG_GetNextSeedRBGCStratum

Return:

- 0 rng is a chain root.
- n The stratum, positive for a chain member.
- BAD_FUNC_ARG rng is null.

```
int wc_RNG_DRBG_GetNextSeedRBGCStratum(
    const WC_RNG * rng
)
```

Report the RBG-chain stratum of rng's banked next seed – race-free via the aperture protocol – for provenance-aware consumers.

Parameters:

- **rng** The RNG object to interrogate.

See:

- wc_RNG_DRBG_GetRBGCStratum
- wc_RNG_DRBG_NextSeedGenerate_RBGC

Return:

- 0 The banked seed has root (source) provenance.
- n The banked seed's stratum, positive for chain provenance.
- BAD_FUNC_ARG rng is null or has no DRBG.
- NOT_READY_E No banked seed is ready.

```
int wc_RNG_DRBG_NextSeedGenerate(
    WC_RNG * rng,
    word32 n
)
```

Bank up to n more bytes of next-seed material from the module's seed source, health-testing and publishing the bank when it completes. The fill is incremental and in-boundary; a scheduling daemon may call this without owning the instance – the single-writer fill and the atomic aperture hand-off make it safe alongside a concurrent consumer.

Parameters:

- **rng** The RNG object whose bank to fill.
- **n** Maximum bytes to bank this call (clamped to space remaining).

See:

- wc_RNG_DRBG_NextSeedNow
- wc_RNG_DRBG_NextSeedCurrent
- wc_RNG_DRBG_NextSeedGenerate_RBGC

Return:

- 0 Bytes were banked (bank may or may not yet be complete).
- ALREADY_E The bank is ready or being consumed.

- ENTROPY_RT_E or ENTROPY_APT_E The SP 800-90B seed health test rejected the banked material, which is burned.
- BAD_FUNC_ARG rng is null or n is 0.
- MISSING_RNG_E rng has no DRBG (RDRAND et al.).

Example

```
// scheduling daemon: fill incrementally until published
int ret = wc_RNG_DRBG_NextSeedGenerate(rng, 16);
if (ret == ALREADY_E) {
    // bank is ready; nothing to do until a consumer claims it
}
```

```
int wc_RNG_DRBG_NextSeedGenerate_RBGC(
    WC_RNG * rng,
    WC_RNG * root,
    word32 n
)
```

The chain-sourced form of wc_RNG_DRBG_NextSeedGenerate(): the banked material is drawn from root's generate function, and the bank is tagged with root's stratum plus one for provenance-aware consumption.

Parameters:

- **rng** The RNG object whose bank to fill.
- **root** The chain parent to draw material from.
- **n** Maximum bytes to bank this call.

See:

- wc_RNG_DRBG_NextSeedGenerate
- wc_RNG_DRBG_GetNextSeedRBGCStratum

Return:

- 0 Bytes were banked.
- ALREADY_E The bank is ready or being consumed.
- ENTROPY_RT_E or ENTROPY_APT_E The SP 800-90B seed health test rejected the banked material, which is burned.
- BAD_FUNC_ARG rng or root is null, or n is 0.
- MISSING_RNG_E rng has no DRBG (RDRAND et al.).
- SEQ_OVERFLOW_E root's stratum is at the representable maximum.

Banking is bound for credited redemption, so the credited no-downgrade rule applies at bank time: a primary-seeded (stratum-0) root is always accepted, and a chained root only when its stratum is strictly less than rng's – banking whose redemption would raise rng's stratum is refused with BAD_FUNC_ARG. The banked material records root's stratum plus one, observable via wc_RNG_DRBG_GetNextSeedRBGCStratum(), and redemption (wc_RNG_DRBG_NextSeedNow()) carries it onto rng.

```
int wc_RNG_DRBG_NextSeedCurrent(
    WC_RNG * rng,
    WC_ATOMIC_INT_ARG * n
)
```

Report the raw next-seed aperture value: a non-negative banked byte count (filling), WC_DRBG_NEXT_SEED_READY, or WC_DRBG_NEXT_SEED_CONSUMING. The snapshot is racy by design; use it for scheduling and diagnostics, not for hand-off decisions.

Parameters:

- **rng** The RNG object to interrogate.
- **n** Receives the aperture value.

See:

- wc_RNG_DRBG_NextSeedGenerate
- wc_RNG_DRBG_NextSeedNow

Return:

- 0 Success
- BAD_FUNC_ARG rng or n is null.
- MISSING_RNG_E rng has no DRBG (RDRAND et al.).

```
int wc_RNG_DRBG_NextSeedNow(
    WC_RNG * rng
)
```

Claim a ready next-seed bank and perform a source-free credited reseed with it – safe in atomic context. The bank empties (use-once) and the reseed counter resets. The caller must own the instance.

Parameters:

- **rng** The RNG object to reseed.

See:

- wc_RNG_DRBG_NextSeedGenerate
- wc_RNG_DRBG_NextSeedNow_Nonce

Return:

- 0 Success
- NOT_READY_E No bank is ready.
- BAD_FUNC_ARG rng is null.
- MISSING_RNG_E rng has no DRBG (RDRAND et al.).
- ENTROPY_RT_E or ENTROPY_APT_E A seed health test failed on the reseed this performs; the instance is condemned.

Example

```
// atomic-context consumer
if (wc_RNG_DRBG_NextSeedNow(rng) == 0) {
    // freshly reseeded without touching the seed source
}
```

```
int wc_RNG_DRBG_NextSeedNow_Nonce(
    WC_RNG * rng,
    const byte * nonce,
    word32 nonceSz
)
```

The nonce-bearing form of wc_RNG_DRBG_NextSeedNow(): the nonce is mixed in as uncredited additional input alongside the banked seed.

Parameters:

- **rng** The RNG object to reseed.
- **nonce** Additional input.
- **nonceSz** Length of nonce in bytes.

See: wc_RNG_DRBG_NextSeedNow

Return:

- 0 Success
- NEEDS_RECOVERY_E A purge crossed the consume (an invalidation epoch boundary): no material is adopted, the entropy-invalidated latch is re-asserted, and a recovery reseed is scheduled.
- NOT_READY_E No bank is ready.
- BAD_FUNC_ARG rng is null, or nonce is null with nonceSz nonzero.
- MISSING_RNG_E rng has no DRBG (RDRAND et al.).
- ENTROPY_RT_E or ENTROPY_APT_E A seed health test failed on the reseed this performs; the instance is condemned.
- DRBG_CONT_FIPS_E The continuous test failed; the DRBG is out of service.
- RNG_FAILURE_E The DRBG is out of service.

```
int wc_RNG_DRBG_NextStirStore(
    WC_RNG * rng,
    const byte * nonce,
    word32 nonceSz
)
```

Bank caller-supplied material (up to WC_RNG_NEXT_STIR_LEN bytes) in the uncredited accumulator beside the banked next seed. Writer-safe without a lease (read-copy-store); if the accumulator is already full, the material is absorbed by xor. Harvested entropy deposited here improves the instance without claiming credit.

Parameters:

- **rng** The RNG object whose accumulator to feed.
- **nonce** Material to bank.
- **nonceSz** Length of nonce in bytes.

See:

- wc_RNG_DRBG_NextStirNow
- wc_RNG_DRBG_Stir

Return:

- 0 Success
- BAD_FUNC_ARG rng or nonce is null, or nonceSz is 0.
- MISSING_RNG_E rng has no DRBG (RDRAND et al.).

```
int wc_RNG_DRBG_NextStirNow(
    WC_RNG * rng
)
```

Stir the banked uncredited accumulator into the DRBG as an uncredited, source-free mix-in – safe in atomic context; the reseed counter is not reset. The caller must own the instance.

Parameters:

- **rng** The RNG object to stir.

See: wc_RNG_DRBG_NextStirStore

Return:

- 0 Success
- NOT_READY_E The accumulator is empty or still accumulating, or the stir is refused (reseed interval, entropy-invalidated quarantine).
- BUSY_E The accumulator was claimed by a racing consumer – the stir is happening by another hand.
- BAD_FUNC_ARG rng is null.
- MISSING_RNG_E rng has no DRBG (RDRAND et al.).
- RNG_FAILURE_E The DRBG is out of service, or its hash failed mid-stir leaving a half-applied update – the instance is then condemned (status DRBG_FAILED).

```
int wc_RNG_lock_get(
    WC_RNG * rng,
    WC_RNG_lock_arg_t extra_bits
)
```

Acquire rng's exclusive-ownership lock latch, spinning on the CAS until acquired, and or the caller's extra bits into the lock word. On an instance without the lock-required policy the call is a successful no-op unless extra bits are supplied.

Parameters:

- **rng** The RNG object to lock.
- **extra_bits** Caller-defined bits (above WC_RNG_LOCK_EXTRA_SHIFT) to set atomically with the acquisition, or 0.

See:

- wc_RNG_lock_put
- wc_RNG_lock_get_conditional
- wc_RNG_lock_read

Return:

- 0 Success
- BAD_FUNC_ARG rng is null.
- BUSY_E The lock is held.
- NEEDS_RECOVERY_E The instance's entropy is invalidated (see wc_RNG_invalidate_entropy()); recover with a credited reseed before use.
- BAD_MUTEX_E (WC_RNG_HAVE_LOCK_FULL_MUTEX) The outer mutex failed.
- UNEXPECTED_STATE_E Spurious acquisition failure; retry.

Example

```
if (wc_RNG_lock_get(rng, 0) == 0) {
    ret = wc_RNG_GenerateBlock(rng, out, sizeof(out));
    wc_RNG_lock_put(rng, 0);
}
```

```
int wc_RNG_lock_get_conditional(
    WC_RNG * rng,
    WC_RNG_lock_arg_t expected_extra_bits,
    WC_RNG_lock_arg_t want_extra_bits
)
```

The conditional form of wc_RNG_lock_get(): acquire only if the current extra bits equal expected_extra_bits, atomically replacing them with want_extra_bits on success. Non-blocking with respect to the condition: a mismatch fails immediately rather than spinning.

Parameters:

- **rng** The RNG object to lock.
- **expected_extra_bits** The extra bits required for acquisition.
- **want_extra_bits** The extra bits to install on acquisition.

See:

- wc_RNG_lock_get
- wc_RNG_lock_put_conditional

Return:

- 0 Success
- BAD_FUNC_ARG rng is null.
- BUSY_E The lock is held, or the extra bits do not match expected_extra_bits.
- NEEDS_RECOVERY_E Entropy-invalidated and WC_RNG_LOCK_ENTROPY_INVALIDATED is not in expected_extra_bits.
- BAD_MUTEX_E (WC_RNG_HAVE_LOCK_FULL_MUTEX) The outer mutex failed.
- UNEXPECTED_STATE_E Spurious acquisition failure; retry.

```
int wc_RNG_lock_put(
    WC_RNG * rng,
    WC_RNG_lock_arg_t extra_bits
)
```

Release rng's lock latch, clearing the supplied extra bits atomically with the release.

Parameters:

- **rng** The RNG object to unlock.
- **extra_bits** Caller-defined bits to clear with the release, or 0.

See:

- wc_RNG_lock_get
- wc_RNG_lock_put_conditional

Return:

- 0 Success
- BAD_FUNC_ARG rng is null.
- OBJECT_NOT_LOCKED_E The latch is not held.
- NEEDS_RECOVERY_E Released successfully; informational notice that the instance is entropy-invalidated.

```
int wc_RNG_lock_put_conditional(
    WC_RNG * rng,
    WC_RNG_lock_arg_t expected_extra_bits,
    WC_RNG_lock_arg_t want_extra_bits
)
```

The conditional form of wc_RNG_lock_put(): release only if the current extra bits equal expected_extra_bits, atomically replacing them with want_extra_bits on success.

Parameters:

- **rng** The RNG object to unlock.

- **expected_extra_bits** The extra bits required for release.
- **want_extra_bits** The extra bits to install on release.

See:

- wc_RNG_lock_put
- wc_RNG_lock_get_conditional

Return:

- 0 Success
- BAD_FUNC_ARG rng is null.
- BUSY_E The extra bits did not match expected_extra_bits.
- OBJECT_NOT_LOCKED_E The latch is not held.
- NEEDS_RECOVERY_E Released successfully; informational notice that the instance is entropy-invalidated.
- UNEXPECTED_STATE_E Spurious release failure.

```
int wc_RNG_lock_read(
    WC_RNG * rng,
    WC_RNG_lock_arg_t * state
)
```

Read rng's lock word: the held/required latch bits, the entropy-invalidated bit, and any caller extra bits. The snapshot is racy by design.

Parameters:

- **rng** The RNG object to interrogate.
- **state** Receives the lock word.

See:

- wc_RNG_lock_get
- wc_RNG_invalidate_entropy

Return:

- 0 Success
- BAD_FUNC_ARG rng or state is null.

```
int wc_RNG_lock_set_extra(
    WC_RNG * rng,
    WC_RNG_lock_arg_t extra_bits
)
```

Atomically set (or) the supplied caller extra bits in rng's lock word. The caller should hold the latch.

Parameters:

- **rng** The RNG object to modify.
- **extra_bits** The bits to set.

See:

- wc_RNG_lock_add_extra
- wc_RNG_lock_clear_extra

Return:

- 0 Success
- BAD_FUNC_ARG rng is null.

```
int wc_RNG_lock_add_extra(
    WC_RNG * rng,
    WC_RNG_lock_arg_t extra_bits
)
```

Atomically add the supplied value to the caller extra-bits field of rng's lock word – for counters carried above WC_RNG_LOCK_EXTRA_SHIFT.

Parameters:

- **rng** The RNG object to modify.
- **extra_bits** The value to add.

See:

- wc_RNG_lock_set_extra
- wc_RNG_lock_clear_extra

Return:

- 0 Success
- BAD_FUNC_ARG rng is null.

```
int wc_RNG_lock_clear_extra(
    WC_RNG * rng,
    WC_RNG_lock_arg_t extra_bits
)
```

Atomically clear the supplied caller extra bits in rng's lock word.

Parameters:

- **rng** The RNG object to modify.
- **extra_bits** The bits to clear.

See:

- wc_RNG_lock_set_extra
- wc_RNG_lock_add_extra

Return:

- 0 Success
- BAD_FUNC_ARG rng is null.

```
int wc_RNG_invalidate_entropy(
    WC_RNG * rng
)
```

Mark rng's seed material untrusted – for VM fork/resume and similar duplication events – opening a new invalidation epoch: banked and pooled pre-event material is purged and wiped first, then the entropy-invalidated bit is latched in the lock word. An invalidated instance refuses service (NEEDS_RECOVERY_E) until recovery-reseeded. Latch or condemn: on any error return the latch is down, and the instance is instead condemned (status DRBG_FAILED). A condemned bank instance is retired and recovered by the entropy daemon; a condemned leaf gets no daemon rescue – its owner sees RNG_FAILURE_E from subsequent operations and recovers it with wc_FreeRng() then wc_InitRng*().

Parameters:

- **rng** The RNG object to invalidate.

See:

- wc_RNG_lock_get
- wc_RNG_register_free_hook

Return:

- 0 Success: purges complete, latch asserted.
- BAD_FUNC_ARG rng is null.
- RNG_FAILURE_E (or other nonzero) A purge or the latch failed; the instance is condemned as above.

Example

```
// VM-resume handler
(void)wc_RNG_invalidate_entropy(rng);
// subsequent wc_RNG_lock_get() returns NEEDS_RECOVERY_E until recovery
```

```
int wc_RNG_register_free_hook(
    WC_RNG * rng,
    wc_RNG_free_hook_cb_t free_hook,
    void * arg
)
```

Register a callback fired by wc_FreeRng() at teardown – for external registries (e.g. a kernel_module registry that must reach every live RNG on a VM duplication event) that need to drop their reference when the object dies.

Parameters:

- **rng** The RNG object to hook.
- **free_hook** The callback.
- **arg** Opaque argument passed to the callback.

See:

- wc_RNG_invalidate_entropy
- wc_FreeRng

Return:

- 0 Success
- BAD_FUNC_ARG rng or free_hook is null.

```
int wc_RNG_Pool_Alloc(
    WC_RNG * rng,
    word32 size
)
```

Attach a random pool to rng: a buffer of size bytes of pre-generated output, filled by wc_RNG_Pool_Collect() and drained atomic-context-safely by wc_RNG_Pool_Extract(). The pool is released with the instance.

Parameters:

- **rng** The RNG object to equip.
- **size** Pool capacity in bytes.

See:

- wc_RNG_Pool_Collect
- wc_RNG_Pool_Extract

Return:

- 0 Success
- BAD_FUNC_ARG rng is null, or size is 0 or out of range.
- MEMORY_E Allocation failed.

- ALREADY_E The pool is already allocated.

Example

```
wc_RNG_Pool_Alloc(rng, 256);
wc_RNG_Pool_Collect(rng, 256);      // sleepable context
word32 n = 16;
if (wc_RNG_Pool_Extract(rng, out, &n) == 0) {
    // n bytes delivered, atomic-context-safe
}
```

```
int wc_RNG_Pool_Collect(
    WC_RNG * rng,
    word32 n
)
```

Generate up to n bytes into rng's pool from rng itself. The collect/extract hand-off is arbitrated by an atomic aperture word, so a single collector is safe alongside concurrent extractors.

Parameters:

- **rng** The RNG object whose pool to fill.
- **n** Maximum bytes to collect this call.

See:

- wc_RNG_Pool_Alloc
- wc_RNG_Pool_Collect2
- wc_RNG_Pool_Extract

Return:

- 0 Success
- ALREADY_E The pool is full or being drained.
- BAD_FUNC_ARG rng is null, has no pool, or n is 0.

```
int wc_RNG_Pool_Collect2(
    WC_RNG * rng_dest,
    WC_RNG * rng_src,
    word32 n
)
```

The two-instance form of wc_RNG_Pool_Collect(): fill rng_dest's pool with output drawn from rng_src – so a service instance's pool can be topped up by a daemon-owned generator.

Parameters:

- **rng_dest** The RNG object whose pool to fill.

- **rng_src** The RNG object to draw output from.
- **n** Maximum bytes to collect this call.

See: wc_RNG_Pool_Collect

Return:

- 0 Success
- ALREADY_E The pool is full or being drained.
- BAD_FUNC_ARG rng_dest or rng_src is null, or rng_dest has no pool, or n is 0.
- BAD_STATE_E The pool is not allocated.
- NOT_READY_E The source could not serve; retry later.

```
int wc_RNG_Pool_Extract(
    WC_RNG * rng,
    byte * out,
    word32 * n
)
```

Drain up to *n bytes from rng's pool into out – atomic-context-safe. On success* n reports the bytes actually delivered; on any error return *n is left unmodified.

Parameters:

- **rng** The RNG object whose pool to drain.
- **out** Receives the output.
- **n** In: bytes requested; out: bytes delivered.

See:

- wc_RNG_Pool_Collect
- wc_RNG_Pool_Current

Return:

- 0 Success
- NOT_READY_E The pool is empty or being filled.
- BAD_FUNC_ARG rng, out, or n is null, or rng has no pool.
- BAD_STATE_E The pool is not allocated.
- RNG_FAILURE_E The instance is out of service.

```
int wc_RNG_Pool_Current(
    WC_RNG * rng,
    word32 * n
)
```

Report the pool's current fill in bytes. The snapshot is racy by design.

Parameters:

- **rng** The RNG object to interrogate.
- **n** Receives the fill.

See: wc_RNG_Pool_Extract

Return:

- 0 Success
- BAD_FUNC_ARG rng or n is null, or rng has no pool.

```
int wc_rng_debug_stats_snap(
    struct wc_rng_debug_stats_snapshot * s,
    const WC_RNG * rng
)
```

Snapshot the global RNG debug counters (WC_RNG_DEBUG_STATS) – seeds and reseeds by provenance, generates, pool and bank traffic – into s, for later delta accounting with wc_rng_debug_stats_sum().

Parameters:

- **s** Receives the snapshot.
- **rng** Optional instance for per-instance context, or null.

See:

- wc_rng_debug_stats_sum
- wc_rng_debug_stats_restore

Return:

- 0 Success
- BAD_FUNC_ARG s is null.

```
int wc_rng_debug_stats_restore(
    const struct wc_rng_debug_stats_snapshot * s,
    WC_RNG * rng
)
```

Restore the global RNG debug counters from a snapshot – so a test can unwind its own accounting.

Parameters:

- **s** The snapshot to restore from.
- **rng** Optional instance for per-instance context, or null.

See: wc_rng_debug_stats_snap

Return:

- 0 Success
- BAD_FUNC_ARG s is null.

```
int wc_rng_debug_stats_sum(
    struct wc_rng_debug_stats_snapshot * s,
    const WC_RNG * rng
)
```

Accumulate the current global RNG debug counters into s – combined with a prior wc_rng_debug_stats_snap(), a delta accounting of the interval's RNG activity.

Parameters:

- **s** The snapshot to accumulate into.
- **rng** Optional instance for per-instance context, or null.

See: wc_rng_debug_stats_snap

Return:

- 0 Success
- BAD_FUNC_ARG s is null.

```
int wc_rng_bank_new(
    struct wc_rng_bank ** ctx,
    int n_rngs,
    word32 flags,
    int timeout_secs,
    void * heap,
    int devId
)
```

Allocate and initialize a bank of n_rngs pre-instantiated WC_RNG instances, checked out and back in by consumers (wc_rng_bank_checkout() et al.). The bank is allocated from heap; release with wc_rng_bank_free(). Bank-level flags (WC_RNG_BANK_FLAG_*) fix the bank's posture at initialization: e.g. _CAN_WAIT admits sleeping, _QUIET suppresses seeding-degradation warnings, _NO_CHECKOUT_REFCOUNTING suppresses per-checkout refcount traffic for container-guaranteed lifetimes, _PREDICTION_RESISTANCE imposes a bank-wide fresh-reseed posture on every sleepable lease.

Parameters:

- **ctx** Receives the allocated bank.
- **n_rngs** Number of instances.
- **flags** Bitwise-or of WC_RNG_BANK_FLAG_* bank-posture flags.

- **timeout_secs** Seeding timeout budget.
- **heap** Heap hint for dynamic allocation.
- **devId** Device id, or INVALID_DEVID.

See:

- wc_rng_bank_init
- wc_rng_bank_checkout
- wc_rng_bank_free

Return:

- 0 Success
- BAD_FUNC_ARG ctx is null or n_rngs is out of range.
- MEMORY_E Allocation failed.
- RNG_FAILURE_E No instance could be seeded within timeout_secs.

Example

```
struct wc_rng_bank *bank = NULL;
if (wc_rng_bank_new(&bank, 4, WC_RNG_BANK_FLAG_CAN_WAIT, 10,
                    NULL, INVALID_DEVID) == 0) {
    // ... checkout/checkin traffic ...
    wc_rng_bank_free(&bank);
}
```

```
int wc_rng_bank_init(
    struct wc_rng_bank * ctx,
    int n_rngs,
    word32 flags,
    int timeout_secs,
    void * heap,
    int devId
)
```

Initialize a caller-provided bank object. Semantics of wc_rng_bank_new(), without the allocation; release with wc_rng_bank_fini().

Parameters:

- **ctx** The bank object to initialize.
- **n_rngs** Number of instances.
- **flags** Bitwise-or of WC_RNG_BANK_FLAG_* bank-posture flags.
- **timeout_secs** Seeding timeout budget.
- **heap** Heap hint for dynamic allocation.
- **devId** Device id, or INVALID_DEVID.

See:

- wc_rng_bank_new

- wc_rng_bank_init_nonce
- wc_rng_bank_fini

Return:

- 0 Success
- BAD_FUNC_ARG ctx is null or n_rngs is out of range.
- BAD_LENGTH_E n_rngs exceeds the static capacity (WC_RNG_BANK_STATIC builds).
- RNG_FAILURE_E No instance could be seeded within timeout_secs.

```
int wc_rng_bank_init_nonce(
    struct wc_rng_bank * ctx,
    int n_rngs,
    word32 flags,
    int timeout_secs,
    void * heap,
    int devId,
    const byte * nonce,
    word32 nonceSz,
    const byte * perso,
    word32 persoSz
)
```

The nonce-bearing form of wc_rng_bank_init(): the nonce is used as additional instantiation input for each instance.

Parameters:

- **ctx** The bank object to initialize.
- **n_rngs** Number of instances.
- **flags** Bitwise-or of WC_RNG_BANK_FLAG_* bank-posture flags.
- **timeout_secs** Seeding timeout budget.
- **heap** Heap hint for dynamic allocation.
- **devId** Device id, or INVALID_DEVID.
- **nonce** Additional instantiation input.
- **nonceSz** Length of nonce in bytes.
- **perso** Optional personalization string (may be null).
- **persoSz** Length of perso in bytes.

See: wc_rng_bank_init

Return:

- 0 Success
- BAD_FUNC_ARG ctx is null, n_rngs is out of range, or nonce is null with nonceSz nonzero.
- RNG_FAILURE_E No instance could be seeded within timeout_secs.
- BAD_LENGTH_E nonceSz exceeds the supported maximum.
- MEMORY_E Allocation failed.
- WC_TIMEOUT_E Instance seeding exceeded timeout_secs.

```
int wc_rng_bank_first_failover_inst_set(
    struct wc_rng_bank * ctx,
    int first_failover_inst
)
```

Designate the first instance of the failover pool: checkouts without a targeted preference rotate through instances at and above first_failover_inst, reserving the lower offsets for targeted (affinity or daemon) use.

Parameters:

- **ctx** The bank to configure.
- **first_failover_inst** The first failover-eligible instance offset.

See: wc_rng_bank_checkout

Return:

- 0 Success
- BAD_FUNC_ARG ctx is null or first_failover_inst is out of range.

```
int wc_rng_bank_set_affinity_handlers(
    struct wc_rng_bank * ctx,
    wc_affinity_lock_fn_t affinity_lock_cb,
    wc_affinity_get_id_fn_t affinity_get_id_cb,
    wc_affinity_unlock_fn_t affinity_unlock_cb,
    void * cb_arg
)
```

Install affinity handlers: callbacks that pin the caller to an execution context (e.g. disable preemption or migration), report its id for instance affinity, and unpin. With handlers installed, WC_RNG_BANK_FLAG_PREFER_AFFINITY_INST checkouts prefer the instance matching the caller's affinity id, and WC_RNG_BANK_FLAG_AFFINITY_LOCK holds the pin across the lease.

Parameters:

- **ctx** The bank to configure.
- **affinity_lock_cb** Pin the caller; may be null.
- **affinity_get_id_cb** Report the caller's affinity id.
- **affinity_unlock_cb** Unpin the caller; may be null.
- **cb_arg** Opaque argument passed to the callbacks.

See: wc_rng_bank_checkout

Return:

- 0 Success
- BAD_FUNC_ARG ctx is null.
- BUSY_E The bank is in service; handlers must be set before use.

```
int wc_rng_bank_fini(
    struct wc_rng_bank * ctx
)
```

Tear down a bank initialized with wc_rng_bank_init(): once the refcount and per-instance lease gates pass, fires any registered free hook and frees every instance. Never waits: a referenced or leased bank is refused with BUSY_E, and the caller quiesces its consumers and retries.

Parameters:

- **ctx** The bank to tear down.

See:

- wc_rng_bank_init
- wc_rng_bank_free
- wc_rng_bank_register_free_hook

Return:

- 0 Success
- BAD_FUNC_ARG ctx is null.
- BUSY_E The bank is still referenced, or an instance lease is outstanding.
- BAD_STATE_E The refcount is below its initialization baseline (teardown of an uninitialized or corrupted bank).

```
int wc_rng_bank_free(
    struct wc_rng_bank ** ctx
)
```

Tear down and release a bank allocated with wc_rng_bank_new().

Parameters:

- **ctx** The bank to release; nulled on success.

See:

- wc_rng_bank_new
- wc_rng_bank_fini

Return:

- 0 Success
- BAD_FUNC_ARG ctx or *ctx is null.
- BAD_STATE_E The bank is still referenced.

```
int wc_rng_bank_default_set(
    struct wc_rng_bank * bank
)
```

Register bank as the process-default bank, retrievable with wc_rng_bank_default_checkout().

Parameters:

- **bank** The bank to register.

See:

- wc_rng_bank_default_checkout
- wc_rng_bank_default_clear

Return:

- 0 Success
- BAD_FUNC_ARG bank is null.
- BAD_STATE_E A default bank is already registered.
- BUSY_E Registration is contended; retry.

```
int wc_rng_bank_default_checkout(
    struct wc_rng_bank ** bank
)
```

Take a reference on the process-default bank.

Parameters:

- **bank** Receives the default bank.

See:

- wc_rng_bank_default_set
- wc_rng_bank_default_checkin

Return:

- 0 Success
- BAD_FUNC_ARG bank is null.
- BAD_STATE_E No default bank is registered.
- NO_DEFAULT_FOUND_E No default bank is registered.

```
int wc_rng_bank_default_checkin(
    struct wc_rng_bank ** bank
)
```

Release a reference taken with wc_rng_bank_default_checkout().

Parameters:

- **bank** The reference to release; nulled on success.

See: wc_rng_bank_default_checkout

Return:

- 0 Success
- BAD_FUNC_ARG bank or *bank is null.

```
int wc_rng_bank_default_clear(
    struct wc_rng_bank * bank
)
```

Unregister the process-default bank.

Parameters:

- **bank** The bank to unregister.

See: wc_rng_bank_default_set

Return:

- 0 Success
- BAD_FUNC_ARG bank is null or is not the registered default.
- BUSY_E Unregistration is contended; retry.

```
int wc_rng_bank_checkout(
    struct wc_rng_bank * bank,
    struct wc_rng_bank_inst ** rng_inst,
    int preferred_inst_offset,
    int timeout_secs,
    word32 flags
)
```

Lease an in-service instance from the bank: either the preferred (or affinity-matched) instance, or – with WC_RNG_BANK_FLAG_CAN_FAIL_OVER_INST – the first available failover instance. On success the caller owns the instance's lock; access the WC_RNG with WC_RNG_BANK_INST_TO_RNG() and return the lease with wc_rng_bank_inst_checkin(). WC_RNG_BANK_FLAG_CONSUME_NEXT_SEED consumes a ready banked next seed in an immediate source-free credited reseed before returning; WC_RNG_BANK_FLAG_PREDICTION_RESISTANCE (per-call, requires _CAN_WAIT) freshly credited-reseeds the lease before the caller's first draw.

Parameters:

- **bank** The bank to lease from.
- **rng_inst** Receives the leased instance.
- **preferred_inst_offset** The preferred instance, or 0.
- **timeout_secs** Wait budget (with _CAN_WAIT).
- **flags** Bitwise-or of WC_RNG_BANK_FLAG_* per-call flags.

See:

- wc_rng_bank_inst_checkin
- wc_rng_bank_recover_inst
- wc_rng_bank_spawn

Return:

- 0 Success; *rng_inst holds the lease.
- BAD_FUNC_ARG bank or rng_inst is null, or the flags are contradictory.
- NEEDS_RECOVERY_E Quarantined: refused without a lease (ordinary checkout), or lease held with recovery owed (_MAYBE_FOR_RECOVERY; see
- BAD_STATE_E (_ERROR_ON_RNG_FAILED) only out-of-service instances were found.
- WC_TIMEOUT_E The wait budget expired.
- RNG_FAILURE_E No serviceable instance.
- BAD_INDEX_E preferred_inst_offset is out of range.
- BUSY_E The selected instance is contended (without _CAN_WAIT).

Out-of-service instances: a targeted (non-failover) checkout admits them; failover checkouts divert around them. WC_RNG_BANK_FLAG_FOR_RECOVERY makes the targeted admission explicit and interaction-safe (recovery machinery and patrols): out-of-service status is expected, the _CONSUME_NEXT_SEED arm is suppressed (a consume would fail on exactly the instances recovery targets), and failover and affinity selection are rejected in combination. WC_RNG_BANK_FLAG_ERROR_ON_RNG_FAILED gives the opposite guarantee: either a lease on an in-service instance, or an error with no lease – never a lease on an out-of-service instance; under _CAN_WAIT an out-of-service instance is retried within the timeout budget (allowing a patrol to restore it), and the distinguished error for a lap or wait that found only out-of-service instances is BAD_STATE_E.

Entropy-invalidated (quarantined) instances refuse ordinary leases with NEEDS_RECOVERY_E, with two admissions. First, when the quarantined instance holds a READY banked next seed, any claimant is admitted and the consume-at-checkout reseed runs unconditionally: the invalidation purge guarantees READY banked material post-dates the invalidation event, so admitting the claimant is completing the recovery. Second, WC_RNG_BANK_FLAG_MAYBE_FOR_RECOVERY admits the caller to a quarantined instance with no banked material, transferring the recovery obligation: checkout then returns NEEDS_RECOVERY_E with the checkout otherwise complete – *rng_inst set; instance lock, affinity locks, and any vector-inhibit state held. This is the robust-mutex (EOWNERDEAD) pattern: an error return with the acquisition complete and persistent, because “this resource needs consistency recovery” is only safely reportable to a caller that already holds it. The caller must either recover the instance (a credited reseed, e.g. wc_RNG_DRBG_Reseed_Now(), clears the quarantine) or check it back in. Ordinary consumers that cannot complete a recovery must not pass this flag.

). *Example*

```
struct wc_rng_bank_inst *inst = NULL;
if (wc_rng_bank_checkout(bank, &inst, 0, 10,
```

```
                        WC_RNG_BANK_FLAG_CAN_WAIT |
                        WC_RNG_BANK_FLAG_CAN_FAIL_OVER_INST) == 0) {
    ret = wc_RNG_GenerateBlock(WC_RNG_BANK_INST_TO_RNG(inst),
                               out, sizeof(out));
    wc_rng_bank_inst_checkin(&inst);
}
```

```
int wc_rng_bank_checkin(
    struct wc_rng_bank * bank,
    struct wc_rng_bank_inst ** rng_inst
)
```

Return a lease through the bank object, validating that rng_inst belongs to bank. Prefer wc_rng_bank_inst_checkin() when only the instance pointer is at hand.

Parameters:

- **bank** The bank the instance belongs to.
- **rng_inst** The lease to return; nulled on success.

See:

- wc_rng_bank_inst_checkin
- wc_rng_bank_checkout

Return:

- 0 Success
- BAD_FUNC_ARG bank or rng_inst is null, or the instance does not belong to bank.
- OBJECT_NOT_LOCKED_E The instance's lease is not held (e.g. a stale duplicate check-in).
- NEEDS_RECOVERY_E Checked in successfully; informational notice that the instance is entropy-invalidated.

```
int wc_rng_bank_inst_checkin(
    struct wc_rng_bank_inst ** rng_inst
)
```

Return a lease by instance pointer alone.

Parameters:

- **rng_inst** The lease to return; nulled on success.

See:

- wc_rng_bank_checkout
- wc_rng_bank_checkin

Return:

- 0 Success
- BAD_FUNC_ARG rng_inst or *rng_inst is null.
- OBJECT_NOT_LOCKED_E The instance's lease is not held.
- NEEDS_RECOVERY_E Checked in successfully; informational notice that the instance is entropy-invalidated.

```
int wc_rng_bank_get_inst_id(
    struct wc_rng_bank_inst * rng_inst
)
```

Report the instance's offset within its bank.

Parameters:

- **rng_inst** The instance to interrogate.

See: wc_rng_bank_checkout

Return:

- n The instance offset, non-negative.
- BAD_FUNC_ARG rng_inst is null.

```
int wc_rng_bank_next_seed_generate(
    struct wc_rng_bank * bank,
    int inst_offset,
    word32 n
)
```

Bank next-seed material for the instance at inst_offset –wc_RNG_DRBG_NextSeedGenerate() through the bank, without taking the instance lock; the daemon-side serialization word arbitrates against concurrent whole-instance reinitialization.

Parameters:

- **bank** The bank.
- **inst_offset** The instance to bank for.
- **n** Maximum bytes to bank this call.

See:

- wc_RNG_DRBG_NextSeedGenerate
- wc_rng_bank_next_seed_generate_rbgc

Return:

- 0 Bytes were banked.
- ALREADY_E The instance's bank is ready or being consumed.
- ENTROPY_RT_E or ENTROPY_APT_E The SP 800-90B seed health test rejected the banked material, which is burned.
- BAD_FUNC_ARG bank is null, inst_offset is out of range, or n is 0.

The daemon-side serialization word (not the instance lock) excludes a concurrent wc_rng_bank_inst_reinit() from freeing the DRBG out from under the gather; lease_holders never consult it, since instance_lock exclusion already covers every lease_holder interaction. The caller must hold a bank reference (e.g. per the daemon association) for the duration of the call. Return taxonomy for a banking rotation: BUSY_E, the gate is held by a reinit – skip this turn; ALREADY_E, sleep until the bank is consumed; MISSING_RNG_E, the instance has no DRBG (RDRAND et al.) and can be retired from the rotation permanently; other errors are transient gather or health-test failures – skip the turn, and alarm if persistent.

```
int wc_rng_bank_next_seed_generate_rbgc(
    struct wc_rng_bank * bank,
    int inst_offset,
    word32 n
)
```

The chain-sourced form of wc_rng_bank_next_seed_generate(): the banked material is drawn from root's generate function and tagged with its provenance.

Parameters:

- **bank** The bank.
- **inst_offset** The instance to bank for.
- **n** Maximum bytes to bank this call.

See:

- wc_rng_bank_next_seed_generate
- wc_RNG_DRBG_NextSeedGenerate_RBGC

Return:

- 0 Bytes were banked.
- ALREADY_E The instance's bank is ready or being consumed.
- ENTROPY_RT_E or ENTROPY_APT_E The SP 800-90B seed health test rejected the banked material, which is burned.
- BAD_FUNC_ARG bank or root is null, inst_offset is out of range, or n is 0.

```
int wc_rng_bank_inst_reinit(
    struct wc_rng_bank * bank,
    struct wc_rng_bank_inst * rng_inst,
    int timeout_secs,
    word32 flags
)
```

Free and reinstantiate a leased instance in place. The caller must hold the lease; the daemon-side serialization word excludes concurrent lockless banking during the cycle.

Parameters:

- **bank** The bank the instance belongs to.
- **rng_inst** The leased instance to reinitialize.
- **timeout_secs** Seeding timeout budget.
- **flags** Bitwise-or of WC_RNG_BANK_FLAG_* flags.

See: wc_rng_bank_recover_inst

Return:

- 0 Success
- BAD_FUNC_ARG bank or rng_inst is null.
- RNG_FAILURE_E Reinstantiation could not be seeded within timeout_secs.
- BUSY_E The whole-instance-operation gate is held (daemon banking in progress); retry.

```
int wc_rng_bank_recover_inst(
    struct wc_rng_bank * bank,
    int inst_offset,
    int timeout_secs,
    word32 flags
)
```

Patrol helper: check out the instance at inst_offset with WC_RNG_BANK_FLAG_FOR_RECOVERY, recover it iff it needs recovery, and check it back in. Two recovery arms: an out-of-service instance (wc_RNG_GetStatus() != WC_DRBG_OK) is reinitialized in place; an in_service but entropy_invalidated (quarantined) instance takes one credited reseed, which clears the quarantine while preserving instance identity. A healthy instance is a success no-op, so callers can invoke this unconditionally on state observed locklessly.

Parameters:

- **bank** The bank.
- **inst_offset** The instance to patrol.
- **timeout_secs** Seeding timeout budget.
- **flags** Bitwise-or of WC_RNG_BANK_FLAG_* flags.

See:

- wc_rng_bank_invalidate_entropy
- wc_rng_bank_checkout

Return:

- 0 Success (recovered, or nothing to recover).
- BAD_FUNC_ARG bank is null, inst_offset is out of range, or the flags are contradictory.
- RNG_FAILURE_E Recovery could not be seeded within timeout_secs.

- BUSY_E The instance lock or whole-instance-operation gate is contended; retry on a later patrol turn.

Example

```
// after a VM duplication event has invalidated the bank:
for (i = 0; i < n_rngs; i++)
    (void)wc_rng_bank_recover_inst(bank, i, 10,
                                   WC_RNG_BANK_FLAG_CAN_WAIT);
```

A stale lockless status observation costs one harmless round trip. BUSY_E reports contention on the instance lock or the whole-instance-operation gate: retry on a later patrol turn. flags may include WC_RNG_BANK_FLAG_CAN_WAIT and WC_RNG_BANK_FLAG_AFFINITY_LOCK, which are passed through.

```
int wc_rng_bank_spawn(
    struct wc_rng_bank * bank,
    WC_RNG * child_rng,
    byte * nonce,
    word32 nonceSz,
    const byte * perso,
    word32 persoSz,
    int preferred_inst_offset,
    int timeout_secs,
    word32 flags
)
```

Spawn an SP 800-90C chain RNG from a bank instance: check out a parent instance (honoring the usual selection flags), instantiate child_rng as its chain child (wc_InitRngNonceRBGC()), and check the parent back in. The child's lifetime is thereafter decoupled from the parent and its bank; release it with wc_FreeRng() (or wc_rng_free() for the heap form, wc_rng_bank_spawn_new()). The child's RBGC stratum is one plus the parent's stratum at instantiation.

Parameters:

- **bank** The bank to spawn from.
- **child_rng** The caller-provided WC_RNG to instantiate.
- **nonce** Optional additional instantiation input.
- **nonceSz** Length of nonce in bytes.
- **perso** Optional personalization string (may be null).
- **persoSz** Length of perso in bytes.
- **preferred_inst_offset** The preferred parent instance, or 0.
- **timeout_secs** Wait budget (with _CAN_WAIT).
- **flags** Bitwise-or of WC_RNG_BANK_FLAG_* per-call flags.

See:

- wc_rng_bank_spawn_new
- wc_InitRngNonceRBGC

Return:

- 0 Success
- BAD_FUNC_ARG bank or child_rng is null, or the flags are contradictory (uncredited or recovery seeding contradict a spawn).
- RNG_FAILURE_E No serviceable parent within the timeout budget.

A recommended nonce, when available, is a racy read of a high-resolution timer (e.g. Linux kernel random_get_entropy()). WC_RNG_BANK_FLAG_CONSUME_NEXT_SEED composes: a ready banked seed is redeemed on the parent before the spawn draw. WC_RNG_BANK_FLAG_STIR and WC_RNG_BANK_FLAG_FOR_RECOVERY are rejected. WC_RNG_BANK_FLAG_ERROR_ON_RNG_FAILED is implied: the parent is guaranteed in-service, or an error is returned with no lease and no child.

```
int wc_rng_bank_spawn_new(
    struct wc_rng_bank * bank,
    WC_RNG ** child_rng,
    byte * nonce,
    word32 nonceSz,
    const byte * perso,
    word32 persoSz,
    int preferred_inst_offset,
    int timeout_secs,
    word32 flags
)
```

The allocating form of wc_rng_bank_spawn(): the child is allocated from the bank's heap and returned through child_rng; release with wc_rng_free().

Parameters:

- **bank** The bank to spawn from.
- **child_rng** Receives the allocated, instantiated WC_RNG.
- **nonce** Optional additional instantiation input.
- **nonceSz** Length of nonce in bytes.
- **perso** Optional personalization string (may be null).
- **persoSz** Length of perso in bytes.
- **preferred_inst_offset** The preferred parent instance, or 0.
- **timeout_secs** Wait budget (with _CAN_WAIT).
- **flags** Bitwise-or of WC_RNG_BANK_FLAG_* per-call flags.

See: wc_rng_bank_spawn

Return:

- 0 Success
- BAD_FUNC_ARG bank or child_rng is null.
- MEMORY_E Allocation failed.
- RNG_FAILURE_E No serviceable parent within the timeout budget.

```
int wc_rng_bank_seed(
    struct wc_rng_bank * bank,
    const byte * seed,
    word32 seedSz,
    const byte * nonce,
    word32 nonceSz,
    int timeout_secs,
    word32 flags
)
```

Reseed every instance with caller-supplied seed material. WC_RNG_BANK_FLAG_STIR mixes the material in without crediting it.

Parameters:

- **bank** The bank to seed.
- **seed** Seed material.
- **seedSz** Length of seed in bytes.
- **nonce** Optional per-instance nonce material (may be null).
- **nonceSz** Length of nonce in bytes.
- **timeout_secs** Wait budget per instance.
- **flags** Bitwise-or of WC_RNG_BANK_FLAG_* flags.

See:

- wc_rng_bank_seed_range
- wc_rng_bank_reseed

Return:

- 0 Success
- BAD_FUNC_ARG bank or seed is null.
- RNG_FAILURE_E An instance could not be reseeded within timeout_secs.

```
int wc_rng_bank_seed_range(
    struct wc_rng_bank * bank,
    int first_inst,
    int last_inst,
    const byte * seed,
    word32 seedSz,
    const byte * nonce,
    word32 nonceSz,
    int timeout_secs,
    word32 flags
)
```

The range form of wc_rng_bank_seed(): seed instances first_inst through last_inst inclusive.

Parameters:

- **bank** The bank to seed.
- **first_inst** The first instance offset.
- **last_inst** The last instance offset.
- **seed** Seed material.
- **seedSz** Length of seed in bytes.
- **nonce** Optional per-instance nonce material (may be null).
- **nonceSz** Length of nonce in bytes.
- **timeout_secs** Wait budget per instance.
- **flags** Bitwise-or of WC_RNG_BANK_FLAG_* flags.

See: wc_rng_bank_seed

Return:

- 0 Success
- BAD_FUNC_ARG bank or seed is null, or the range is out of bounds.
- RNG_FAILURE_E An instance could not be reseeded within timeout_secs.
- BAD_INDEX_E The instance range is invalid.
- BAD_STATE_E The bank is not initialized.
- NO_DEFAULT_FOUND_E bank is null and no default bank is registered.

```
int wc_rng_bank_reseed(
    struct wc_rng_bank * bank,
    const byte * nonce,
    word32 nonceSz,
    int timeout_secs,
    word32 flags
)
```

Reseed every instance from the module's seed source.

Parameters:

- **bank** The bank to reseed.
- **nonce** Optional per-instance nonce material (may be null).
- **nonceSz** Length of nonce in bytes.
- **timeout_secs** Wait budget per instance.
- **flags** Bitwise-or of WC_RNG_BANK_FLAG_* flags.

See:

- wc_rng_bank_reseed_range
- wc_rng_bank_seed

Return:

- 0 Success
- BAD_FUNC_ARG bank is null.
- RNG_FAILURE_E An instance could not be reseeded within timeout_secs.

```
int wc_rng_bank_reseed_range(
    struct wc_rng_bank * bank,
    int first_inst,
    int last_inst,
    const byte * nonce,
    word32 nonceSz,
    int timeout_secs,
    word32 flags
)
```

The range form of wc_rng_bank_reseed().

Parameters:

- **bank** The bank to reseed.
- **nonce** Optional per-instance nonce material (may be null).
- **nonceSz** Length of nonce in bytes.
- **first_inst** The first instance offset.
- **last_inst** The last instance offset.
- **timeout_secs** Wait budget per instance.
- **flags** Bitwise-or of WC_RNG_BANK_FLAG_* flags.

See: wc_rng_bank_reseed

Return:

- 0 Success
- BAD_FUNC_ARG bank is null, or the range is out of bounds.
- RNG_FAILURE_E An instance could not be reseeded within timeout_secs.
- BAD_INDEX_E The instance range is invalid.
- BAD_STATE_E The bank is not initialized.
- WC_TIMEOUT_E The walk exceeded timeout_secs.

```
int wc_rng_bank_invalidate_entropy(
    struct wc_rng_bank * bank,
    word32 flags
)
```

Set the entropy-invalidated latch on every instance (see wc_RNG_invalidate_entropy()): cached entropy products are discarded, and each instance is forced through a credited reseed before its next generate serves output. Lock_free and constant_time per instance; safe from a state_invalidation event context (VM fork/resume).

Parameters:

- **bank** The bank to invalidate.
- **flags** Bitwise-or of WC_RNG_BANK_FLAG_* flags.

See:

- wc_RNG_invalidate_entropy

- wc_rng_bank_recover_inst

Return:

- 0 Success
- BAD_FUNC_ARG bank is null.
- BAD_STATE_E The bank is not initialized.

Example

```
// VM-resume notifier
(void)wc_rng_bank_invalidate_entropy(bank, WC_RNG_BANK_FLAG_NONE);
// instances recover on next checkout, or by patrol:
//   wc_rng_bank_recover_inst()
```

Also purges each instance's banked next-seed apertures: the purge is the provenance guarantee that a READY bank observed after the event holds post-event material (see wc_rng_bank_checkout()'s recovery admissions). Walks every instance even on error, returning the first error. flags must be 0.

```
int wc_rng_bank_daemon_reserve(
    struct wc_rng_bank * bank,
    WC_ATOMIC_UINT_ARG magic
)
```

Reserve the bank's daemon slot with a caller-chosen nonzero magic word, admitting exactly one scheduling daemon per bank. The lifecycle is strictly ordered: _reserve, then _register, then _unregister, then _release.

Parameters:

- **bank** The bank to claim.
- **magic** The daemon's magic word.

See:

- wc_rng_bank_daemon_register
- wc_rng_bank_daemon_release

Return:

- 0 Success
- BAD_FUNC_ARG bank is null or magic is the free sentinel.
- BUSY_E The slot is claimed, or claiming is contended.

```
int wc_rng_bank_daemon_register(
    struct wc_rng_bank * bank,
    void * daemon,
    WC_ATOMIC_UINT_ARG magic
)
```

Register the daemon object in a slot reserved with the same magic word.

Parameters:

- **bank** The bank.
- **daemon** The daemon object to register.
- **magic** The daemon's magic word.

See:

- wc_rng_bank_daemon_reserve
- wc_rng_bank_daemon_unregister

Return:

- 0 Success
- BAD_FUNC_ARG bank or daemon is null.
- ALREADY_E The slot is already in the requested state.
- WRONG_TYPE_OBJECT_E magic does not match the claim.
- BUSY_E The slot transition is contended; retry.

```
int wc_rng_bank_daemon_unregister(
    struct wc_rng_bank * bank,
    void ** daemon,
    WC_ATOMIC_UINT_ARG magic
)
```

Unregister the daemon object, returning it through daemon.

Parameters:

- **bank** The bank.
- **daemon** Receives the registered daemon object.
- **magic** The daemon's magic word.

See:

- wc_rng_bank_daemon_register
- wc_rng_bank_daemon_release

Return:

- 0 Success
- BAD_FUNC_ARG bank or daemon is null.
- ALREADY_E The slot is already in the requested state.
- WRONG_TYPE_OBJECT_E magic does not match the claim.
- BUSY_E The slot transition is contended; retry.

```
int wc_rng_bank_daemon_release(
    struct wc_rng_bank * bank,
    WC_ATOMIC_UINT_ARG magic
)
```

Release the daemon slot claimed with magic, returning it to the free sentinel.

Parameters:

- **bank** The bank.
- **magic** The daemon's magic word.

See: wc_rng_bank_daemon_reserve

Return:

- 0 Success
- BAD_FUNC_ARG bank is null.
- ALREADY_E The slot is already in the requested state.
- WRONG_TYPE_OBJECT_E magic does not match the claim.
- BUSY_E The slot transition is contended; retry.

```
int wc_rng_bank_register_free_hook(
    struct wc_rng_bank * bank,
    wc_rng_bank_free_hook_cb_t free_hook,
    void * arg
)
```

Register a callback fired by wc_rng_bank_fini() once its refcount and leak gates pass – i.e. once teardown is committed – for external registries that must drop their reference when the bank dies. One-shot: the hook is cleared before firing. A null free_hook unregisters.

Parameters:

- **bank** The bank to hook.
- **free_hook** The callback.
- **arg** Opaque argument passed to the callback.

See:

- wc_rng_bank_fini
- wc_RNG_register_free_hook

Return:

- 0 Success
- BAD_FUNC_ARG bank is null.

```
int wc_InitRng_BankRef(
    struct wc_rng_bank * bank,
    WC_RNG * rng
)
```

Initialize rng as a bank reference: a WC_RNG with no DRBG of its own, whose wc_RNG_GenerateBlock() transparently checks an instance out of bank, generates, and checks it back in. Release with wc_FreeRng().

Parameters:

- **bank** The bank to reference.
- **rng** The WC_RNG to initialize as a reference.

See:

- wc_BankRef_Release
- wc_rng_new_bankref

Return:

- 0 Success
- BAD_FUNC_ARG bank or rng is null.

Example

```
WC_RNG rng;
if (wc_InitRng_BankRef(bank, &rng) == 0) {
    // rng now serves through the bank
    ret = wc_RNG_GenerateBlock(&rng, out, sizeof(out));
    wc_FreeRng(&rng);
}
```

```
int wc_BankRef_Release(
    WC_RNG * rng
)
```

Release a bank reference. wc_FreeRng() calls this automatically for bank references; direct use is rarely needed.

Parameters:

- **rng** The bank reference to release.

See: wc_InitRng_BankRef

Return:

- 0 Success
- BAD_FUNC_ARG rng is null or is not a bank reference.

```
int wc_rng_new_bankref(
    struct wc_rng_bank * bank,
    WC_RNG ** rng
)
```

The allocating form of wc_InitRng_BankRef(): the reference is allocated from the bank's heap; release with wc_rng_free().

Parameters:

- **bank** The bank to reference.
- **rng** Receives the allocated bank reference.

See: wc_InitRng_BankRef

Return:

- 0 Success
- BAD_FUNC_ARG bank or rng is null.
- MEMORY_E Allocation failed.

```
int wc_rng_bank_debug_stats_snap(
    struct wc_rng_debug_stats_snapshot * s,
    struct wc_rng_bank * bank
)
```

Snapshot the RNG debug counters (WC_RNG_DEBUG_STATS) with bank-level context.

Parameters:

- **s** Receives the snapshot.
- **bank** Optional bank for context, or null.

See: wc_rng_debug_stats_snap

Return:

- 0 Success
- BAD_FUNC_ARG s is null.

B.9 Signature API

B.8.3.104 function wc_rng_bank_debug_stats_snap

B.9.1 Functions

	Name
int	**wc_SignatureGetSize**(enum wc_SignatureType sig_type, const void * key, word32 key_len)This function returns the maximum size of the resulting signature.
int	**wc_SignatureVerify**(enum wc_HashType hash_type, enum wc_SignatureType sig_type, const byte * data, word32 data_len, const byte * sig, word32 sig_len, void * key, word32 key_len)This function validates a signature by hashing the data and using the resulting hash and key to verify the signature.
int	**wc_SignatureGenerate**(enum wc_HashType hash_type, enum wc_SignatureType sig_type, const byte * data, word32 data_len, byte * sig, word32 * sig_len, void * key, word32 key_len, WC_RNG * rng)This function generates a signature from the data using a key. It first creates a hash of the data then signs the hash using the key.
int	**wc_SignatureVerifyHash**(enum wc_HashType hash_type, enum wc_SignatureType sig_type, const byte * hash_data, word32 hash_len, const byte * sig, word32 sig_len, void * key, word32 key_len)This function verifies a signature using a pre-computed hash. Unlike wc_SignatureVerify which hashes the data first, this function takes the hash directly and verifies the signature against it. If sig_type is WC_SIGNATURE_TYPE_RSA_W_ENC, hash data must be encoded with wc_EncodeSignature prior to calling.
int	**wc_SignatureGenerateHash**(enum wc_HashType hash_type, enum wc_SignatureType sig_type, const byte * hash_data, word32 hash_len, byte * sig, word32 * sig_len, void * key, word32 key_len, WC_RNG * rng)This function generates a signature from a pre-computed hash. Unlike wc_SignatureGenerate which hashes the data first, this function takes the hash directly and signs it. If sig_type is WC_SIGNATURE_TYPE_RSA_W_ENC, hash data must be encoded with wc_EncodeSignature prior to calling.

	Name
int	**wc_SignatureGenerateHash_ex**(enum wc_HashType hash_type, enum wc_SignatureType sig_type, const byte * hash_data, word32 hash_len, byte * sig, word32 * sig_len, void * key, word32 key_len, WC_RNG * rng, int verify)This function generates a signature from a pre-computed hash with extended options. This is similar to wc_SignatureGenerateHash but allows optional verification of the signature after generation.
int	**wc_SignatureGenerate_ex**(enum wc_HashType hash_type, enum wc_SignatureType sig_type, const byte * data, word32 data_len, byte * sig, word32 * sig_len, void * key, word32 key_len, WC_RNG * rng, int verify)This function generates a signature from data with extended options. This is similar to wc_SignatureGenerate but allows optional verification of the signature after generation.

B.9.2 Functions Documentation

```
int wc_SignatureGetSize(
    enum wc_SignatureType sig_type,
    const void * key,
    word32 key_len
)
```

This function returns the maximum size of the resulting signature.

Parameters:

- **sig_type** A signature type enum value such as WC_SIGNATURE_TYPE_ECC or WC_SIGNATURE_TYPE_RSA.
- **key** Pointer to the key structure corresponding to sig_type: pass an ecc_key* for WC_SIGNATURE_TYPE_ECC, or a RsaKey* for WC_SIGNATURE_TYPE_RSA / WC_SIGNATURE_TYPE_RSA_W_ENC. The caller is responsible for ensuring the pointer refers to the correct type; this function cannot verify the actual runtime type of the object.
- **key_len** If key is non-NULL, key_len must be exactly sizeof(ecc_key) or sizeof(RsaKey) matching the sig_type. Passing any other value causes the function to return BAD_FUNC_ARG without dereferencing key. Always pass the size of the concrete key type at the call site: if you have a typed pointer (e.g., ecc_key* k), use sizeof(k); otherwise use sizeof(ecc_key) or sizeof(RsaKey) directly. Do not use sizeof(*key) on the const void parameter itself, as dereferencing void is invalid.

See:

- wc_HashGetDigestSize
- wc_SignatureGenerate
- wc_SignatureVerify

Return: Returns SIG_TYPE_E if sig_type is not supported. Returns BAD_FUNC_ARG if sig_type was invalid or key_len does not exactly match the size of the expected key structure. A positive return value indicates the maximum size of a signature.

Example

```
// Get signature length
enum wc_SignatureType sig_type = WC_SIGNATURE_TYPE_ECC;
ecc_key eccKey;
word32 sigLen;
wc_ecc_init(&eccKey);
sigLen = wc_SignatureGetSize(sig_type, &eccKey, sizeof(eccKey));
if (sigLen > 0) {
    // Success
}
```

```
int wc_SignatureVerify(
    enum wc_HashType hash_type,
    enum wc_SignatureType sig_type,
    const byte * data,
    word32 data_len,
    const byte * sig,
    word32 sig_len,
    void * key,
    word32 key_len
)
```

This function validates a signature by hashing the data and using the resulting hash and key to verify the signature.

Parameters:

- **hash_type** A hash type from the “enum wc_HashType” such as “WC_HASH_TYPE_SHA256”.
- **sig_type** A signature type enum value such as WC_SIGNATURE_TYPE_ECC or WC_SIGNATURE_TYPE_RSA.
- **data** Pointer to buffer containing the data to hash.
- **data_len** Length of the data buffer.
- **sig** Pointer to buffer to output signature.
- **sig_len** Length of the signature output buffer.
- **key** Pointer to the key structure corresponding to sig_type. See wc_SignatureGetSize() for the type-safety constraints that apply to this parameter.
- **key_len** Must be exactly sizeof(ecc_key) or sizeof(RsaKey) matching sig_type. See wc_SignatureGetSize().

See:

- wc_SignatureGetSize
- wc_SignatureGenerate

Return:

- 0 Success
- SIG_TYPE_E -231, signature type not enabled/ available
- BAD_FUNC_ARG -173, bad function argument provided
- BUFFER_E -132, output buffer too small or input too large.

Example

```
int ret;
ecc_key eccKey;

// Import the public key
wc_ecc_init(&eccKey);
ret = wc_ecc_import_x963(eccPubKeyBuf, eccPubKeyLen, &eccKey);
// Perform signature verification using public key
ret = wc_SignatureVerify(
WC_HASH_TYPE_SHA256, WC_SIGNATURE_TYPE_ECC,
fileBuf, fileLen,
sigBuf, sigLen,
&eccKey, sizeof(eccKey));
printf("Signature Verification: %s
(%d)\n", (ret == 0) ? "Pass" : "Fail", ret);
wc_ecc_free(&eccKey);
```

```
int wc_SignatureGenerate(
    enum wc_HashType hash_type,
    enum wc_SignatureType sig_type,
    const byte * data,
    word32 data_len,
    byte * sig,
    word32 * sig_len,
    void * key,
    word32 key_len,
    WC_RNG * rng
)
```

This function generates a signature from the data using a key. It first creates a hash of the data then signs the hash using the key.

Parameters:

- **hash_type** A hash type from the "enum wc_HashType" such as "WC_HASH_TYPE_SHA256".
- **sig_type** A signature type enum value such as WC_SIGNATURE_TYPE_ECC or WC_SIGNATURE_TYPE_RSA.
- **data** Pointer to buffer containing the data to hash.
- **data_len** Length of the data buffer.
- **sig** Pointer to buffer to output signature.
- **sig_len** Length of the signature output buffer.
- **key** Pointer to the key structure corresponding to sig_type. See wc_SignatureGetSize() for the type-safety constraints that apply to this parameter.
- **key_len** Must be exactly sizeof(ecc_key) or sizeof(RsaKey) matching sig_type. See wc_SignatureGetSize().
- **rng** Pointer to an initialized RNG structure.

See:

- wc_SignatureGetSize
- wc_SignatureVerify

Return:

- 0 Success
- SIG_TYPE_E -231, signature type not enabled/ available
- BAD_FUNC_ARG -173, bad function argument provided
- BUFFER_E -132, output buffer too small or input too large.

Example

```
int ret;
WC_RNG rng;
ecc_key eccKey;

wc_InitRng(&rng);
wc_ecc_init(&eccKey);

// Generate key
ret = wc_ecc_make_key(&rng, 32, &eccKey);

// Get signature length and allocate buffer
sigLen = wc_SignatureGetSize(sig_type, &eccKey, sizeof(eccKey));
sigBuf = malloc(sigLen);

// Perform signature verification using public key
ret = wc_SignatureGenerate(
    WC_HASH_TYPE_SHA256, WC_SIGNATURE_TYPE_ECC,
    fileBuf, fileLen,
    sigBuf, &sigLen,
    &eccKey, sizeof(eccKey),
    &rng);
printf("Signature Generation: %s
(%d)\n", (ret == 0) ? "Pass" : "Fail", ret);

free(sigBuf);
wc_ecc_free(&eccKey);
wc_FreeRng(&rng);
```

```
int wc_SignatureVerifyHash(
    enum wc_HashType hash_type,
    enum wc_SignatureType sig_type,
    const byte * hash_data,
    word32 hash_len,
    const byte * sig,
    word32 sig_len,
    void * key,
    word32 key_len
)
```

This function verifies a signature using a pre-computed hash. Unlike wc_SignatureVerify which hashes the data first, this function takes the hash directly and verifies the signature against it. If sig_type

is WC_SIGNATURE_TYPE_RSA_W_ENC, hash data must be encoded with wc_EncodeSignature prior to calling.

Parameters:

- **hash_type** A hash type from enum wc_HashType
- **sig_type** A signature type such as WC_SIGNATURE_TYPE_ECC or WC_SIGNATURE_TYPE_RSA
- **hash_data** Pointer to buffer containing the hash to verify
- **hash_len** Length of the hash buffer
- **sig** Pointer to buffer containing the signature
- **sig_len** Length of the signature buffer
- **key** Pointer to the key structure corresponding to sig_type. See wc_SignatureGetSize() for the type-safety constraints that apply to this parameter.
- **key_len** Must be exactly sizeof(ecc_key) or sizeof(RsaKey) matching sig_type. See wc_SignatureGetSize().

See:

- wc_SignatureVerify
- wc_SignatureGenerateHash

Return:

- 0 Success
- SIG_TYPE_E Signature type not enabled/available
- BAD_FUNC_ARG Bad function argument provided
- BUFFER_E Output buffer too small or input too large

Example

```
ecc_key eccKey;
byte hash[WC_SHA256_DIGEST_SIZE];
byte sig[ECC_MAX_SIG_SIZE];
word32 sigLen = sizeof(sig);

wc_ecc_init(&eccKey);
// import public key, signature, and pre-computed hash ...
int ret = wc_SignatureVerifyHash(WC_HASH_TYPE_SHA256,
                                 WC_SIGNATURE_TYPE_ECC, hash,
                                 sizeof(hash), sig, sigLen,
                                 &eccKey, sizeof(eccKey));
if (ret == 0) {
    // signature verified
}
```

```
int wc_SignatureGenerateHash(
    enum wc_HashType hash_type,
    enum wc_SignatureType sig_type,
    const byte * hash_data,
    word32 hash_len,
    byte * sig,
```

```
    word32 * sig_len,
    void * key,
    word32 key_len,
    WC_RNG * rng
)
```

This function generates a signature from a pre-computed hash. Unlike wc_SignatureGenerate which hashes the data first, this function takes the hash directly and signs it. If sig_type is WC_SIGNATURE_TYPE_RSA_W_ENC, hash data must be encoded with wc_EncodeSignature prior to calling.

Parameters:

- **hash_type** A hash type from enum wc_HashType
- **sig_type** A signature type such as WC_SIGNATURE_TYPE_ECC or WC_SIGNATURE_TYPE_RSA
- **hash_data** Pointer to buffer containing the hash to sign
- **hash_len** Length of the hash buffer
- **sig** Pointer to buffer to output signature
- **sig_len** Pointer to length of signature output buffer
- **key** Pointer to the key structure corresponding to sig_type. See wc_SignatureGetSize() for the type-safety constraints that apply to this parameter.
- **key_len** Must be exactly sizeof(ecc_key) or sizeof(RsaKey) matching sig_type. See wc_SignatureGetSize().
- **rng** Pointer to an initialized RNG structure

See:

- wc_SignatureGenerate
- wc_SignatureVerifyHash

Return:

- 0 Success
- SIG_TYPE_E Signature type not enabled/available
- BAD_FUNC_ARG Bad function argument provided
- BUFFER_E Output buffer too small or input too large

Example

```
WC_RNG rng;
ecc_key eccKey;
byte hash[WC_SHA256_DIGEST_SIZE];
byte sig[ECC_MAX_SIG_SIZE];
word32 sigLen = sizeof(sig);

wc_InitRng(&rng);
wc_ecc_init(&eccKey);
wc_ecc_make_key(&rng, 32, &eccKey);
// generate signature from pre-computed hash
int ret = wc_SignatureGenerateHash(WC_HASH_TYPE_SHA256,
                                   WC_SIGNATURE_TYPE_ECC, hash,
                                   sizeof(hash), sig, &sigLen,
                                   &eccKey, sizeof(eccKey), &rng);
```

```
int wc_SignatureGenerateHash_ex(
    enum wc_HashType hash_type,
    enum wc_SignatureType sig_type,
    const byte * hash_data,
    word32 hash_len,
    byte * sig,
    word32 * sig_len,
    void * key,
    word32 key_len,
    WC_RNG * rng,
    int verify
)
```

This function generates a signature from a pre-computed hash with extended options. This is similar to wc_SignatureGenerateHash but allows optional verification of the signature after generation.

Parameters:

- **hash_type** A hash type from enum wc_HashType
- **sig_type** A signature type such as WC_SIGNATURE_TYPE_ECC or WC_SIGNATURE_TYPE_RSA
- **hash_data** Pointer to buffer containing the hash to sign
- **hash_len** Length of the hash buffer
- **sig** Pointer to buffer to output signature
- **sig_len** Pointer to length of signature output buffer
- **key** Pointer to the key structure corresponding to sig_type. See wc_SignatureGetSize() for the type-safety constraints that apply to this parameter.
- **key_len** Must be exactly sizeof(ecc_key) or sizeof(RsaKey) matching sig_type. See wc_SignatureGetSize().
- **rng** Pointer to an initialized RNG structure
- **verify** If non-zero, verify the signature after generation

See:

- wc_SignatureGenerateHash
- wc_SignatureGenerate_ex

Return:

- 0 Success
- SIG_TYPE_E Signature type not enabled/available
- BAD_FUNC_ARG Bad function argument provided
- BUFFER_E Output buffer too small or input too large

Example

```
WC_RNG rng;
ecc_key eccKey;
byte hash[WC_SHA256_DIGEST_SIZE];
byte sig[ECC_MAX_SIG_SIZE];
word32 sigLen = sizeof(sig);
```

```
wc_InitRng(&rng);
wc_ecc_init(&eccKey);
wc_ecc_make_key(&rng, 32, &eccKey);
int ret = wc_SignatureGenerateHash_ex(WC_HASH_TYPE_SHA256,
                                      WC_SIGNATURE_TYPE_ECC, hash,
                                      sizeof(hash), sig, &sigLen,
                                      &eccKey, sizeof(eccKey),
                                      &rng, 1);
```

```
int wc_SignatureGenerate_ex(
    enum wc_HashType hash_type,
    enum wc_SignatureType sig_type,
    const byte * data,
    word32 data_len,
    byte * sig,
    word32 * sig_len,
    void * key,
    word32 key_len,
    WC_RNG * rng,
    int verify
)
```

This function generates a signature from data with extended options. This is similar to wc_SignatureGenerate but allows optional verification of the signature after generation.

Parameters:

- **hash_type** A hash type from enum wc_HashType
- **sig_type** A signature type such as WC_SIGNATURE_TYPE_ECC or WC_SIGNATURE_TYPE_RSA
- **data** Pointer to buffer containing the data to hash and sign
- **data_len** Length of the data buffer
- **sig** Pointer to buffer to output signature
- **sig_len** Pointer to length of signature output buffer
- **key** Pointer to the key structure corresponding to sig_type. See wc_SignatureGetSize() for the type-safety constraints that apply to this parameter.
- **key_len** Must be exactly sizeof(ecc_key) or sizeof(RsaKey) matching sig_type. See wc_SignatureGetSize().
- **rng** Pointer to an initialized RNG structure
- **verify** If non-zero, verify the signature after generation

See:

- wc_SignatureGenerate
- wc_SignatureGenerateHash_ex

Return:

- 0 Success
- SIG_TYPE_E Signature type not enabled/available
- BAD_FUNC_ARG Bad function argument provided
- BUFFER_E Output buffer too small or input too large

Example

```
WC_RNG rng;
ecc_key eccKey;
byte data[]; // data to sign
byte sig[ECC_MAX_SIG_SIZE];
word32 sigLen = sizeof(sig);

wc_InitRng(&rng);
wc_ecc_init(&eccKey);
wc_ecc_make_key(&rng, 32, &eccKey);
int ret = wc_SignatureGenerate_ex(WC_HASH_TYPE_SHA256,
                                  WC_SIGNATURE_TYPE_ECC, data,
                                  sizeof(data), sig, &sigLen,
                                  &eccKey, sizeof(eccKey), &rng, 1);
```

B.10 wolfCrypt Init and Cleanup

B.9.2.7 function wc_SignatureGenerate_ex

B.10.1 Functions

	Name
int	**wc_HashGetOID**(enum wc_HashType hash_type)This function will return the OID for the wc_HashType provided.
int	**wc_HashGetDigestSize**(enum wc_HashType hash_type)This function returns the size of the digest (output) for a hash_type. The returns size is used to make sure the output buffer provided to wc_Hash is large enough.
int	**wc_Hash**(enum wc_HashType hash_type, const byte * data, word32 data_len, byte * hash, word32 hash_len)This function performs a hash on the provided data buffer and returns it in the hash buffer provided.
int	**wc_HashUpdate**(wc_HashAlg * hash, enum wc_HashType type, const byte * data, word32 dataSz)This function updates the hash in a wc_HashAlg object with the provided data. The hash algorithm used is the one stored in the wc_HashAlg object when it was initialized with wc_HashInit() or wc_HashInit_ex().
int	**wc_HashFinal**(wc_HashAlg * hash, enum wc_HashType type, byte * out)This function finalizes the hash in a wc_HashAlg object and writes the resulting digest to out. The hash algorithm used is the one stored in the wc_HashAlg object when it was initialized with wc_HashInit() or wc_HashInit_ex().

	Name
int	**wc_HashFree**(wc_HashAlg * hash, enum wc_HashType type)This function frees any resources held by a wc_HashAlg object. The hash algorithm used is the one stored in the wc_HashAlg object when it was initialized with wc_HashInit() or wc_HashInit_ex(). The wc_HashAlg object itself is not freed.
int	**wc_HashSetFlags**(wc_HashAlg * hash, enum wc_HashType type, word32 flags)This function sets the flags of the underlying hash object in a wc_HashAlg object. The hash algorithm used is the one stored in the wc_HashAlg object when it was initialized with wc_HashInit() or wc_HashInit_ex(). Available when WOLFSSL_HASH_FLAGS is defined.
int	**wc_HashGetFlags**(wc_HashAlg * hash, enum wc_HashType type, word32 * flags)This function gets the flags of the underlying hash object in a wc_HashAlg object. The hash algorithm used is the one stored in the wc_HashAlg object when it was initialized with wc_HashInit() or wc_HashInit_ex(). Available when WOLFSSL_HASH_FLAGS is defined.
int	**wc_UnpackOctets**(byte * out, word32 outSz, const byte * in, word32 inSz, word32 octetSz)Expands a packed octet stream into the one_octet_per_byte_cell form every wolfCrypt byte* API expects. Declared only when WOLFSSL_WIDE_BYTE is set (CHAR_BIT != 8); elsewhere the two layouts coincide and no conversion is needed.
int	**wc_PackOctets**(byte * out, word32 outSz, const byte * in, word32 inSz, word32 octetSz)Packs a one_octet_per_byte_cell buffer into an octet stream, for storing to flash or handing to a byte_oriented peripheral. Inverse of wc_UnpackOctets(); requires WOLFSSL_WIDE_BYTE.
int	**wolfCrypt_Init**(void)Used to initialize resources used by wolfCrypt.
int	**wolfCrypt_Cleanup**(void)Used to clean up resources used by wolfCrypt.

B.10.2 Functions Documentation

```
int wc_HashGetOID(
    enum wc_HashType hash_type
)
```

This function will return the OID for the wc_HashType provided.

Parameters:

- **hash_type** A hash type from the “enum wc_HashType” such as “WC_HASH_TYPE_SHA256”.

See:

- wc_HashGetDigestSize
- wc_Hash

Return:

- OID returns value greater than 0
- HASH_TYPE_E hash type not supported.
- BAD_FUNC_ARG one of the provided arguments is incorrect.

Example

```
enum wc_HashType hash_type = WC_HASH_TYPE_SHA256;
int oid = wc_HashGetOID(hash_type);
if (oid > 0) {
    // Success
}
```

```
int wc_HashGetDigestSize(
    enum wc_HashType hash_type
)
```

This function returns the size of the digest (output) for a hash_type. The returns size is used to make sure the output buffer provided to wc_Hash is large enough.

Parameters:

- **hash_type** A hash type from the “enum wc_HashType” such as “WC_HASH_TYPE_SHA256”.

See: wc_Hash

Return:

- Success A positive return value indicates the digest size for the hash.
- Error Returns HASH_TYPE_E if hash_type is not supported.
- Failure Returns BAD_FUNC_ARG if an invalid hash_type was used.

Example

```
int hash_len = wc_HashGetDigestSize(hash_type);
if (hash_len <= 0) {
WOLFSSL_MSG("Invalid hash type/len");
return BAD_FUNC_ARG;
}
```

```
int wc_Hash(
    enum wc_HashType hash_type,
    const byte * data,
    word32 data_len,
    byte * hash,
    word32 hash_len
)
```

This function performs a hash on the provided data buffer and returns it in the hash buffer provided.

Parameters:

- **hash_type** A hash type from the "enum wc_HashType" such as "WC_HASH_TYPE_SHA256".
- **data** Pointer to buffer containing the data to hash.
- **data_len** Length of the data buffer.
- **hash** Pointer to buffer used to output the final hash to.
- **hash_len** Length of the hash buffer.

See: wc_HashGetDigestSize

Return: 0 Success, else error (such as BAD_FUNC_ARG or BUFFER_E).

Example

```
enum wc_HashType hash_type = WC_HASH_TYPE_SHA256;
int hash_len = wc_HashGetDigestSize(hash_type);
if (hash_len > 0) {
    int ret = wc_Hash(hash_type, data, data_len, hash_data, hash_len);
    if(ret == 0) {
        // Success
    }
}
```

```
int wc_HashUpdate(
    wc_HashAlg * hash,
    enum wc_HashType type,
    const byte * data,
    word32 dataSz
)
```

This function updates the hash in a wc_HashAlg object with the provided data. The hash algorithm used is the one stored in the wc_HashAlg object when it was initialized with wc_HashInit() or wc_HashInit_ex().

Parameters:

- **hash** Pointer to the wc_HashAlg object, previously initialized with wc_HashInit() or wc_HashInit_ex().
- **type** Must be either WC_HASH_TYPE_NONE or the same hash type that was used to initialize the hash object. The hash type stored in the hash object during initialization is always used to select the algorithm. Any other value results in BAD_FUNC_ARG.
- **data** Pointer to buffer containing the data to hash. May be NULL only if dataSz is 0.
- **dataSz** Length of the data buffer in bytes.

See:

- wc_HashFinal
- wc_HashFree

Return:

- 0 Success.
- BAD_FUNC_ARG hash is NULL, data is NULL while dataSz is greater than 0, the hash object was not initialized with a valid hash type, or type is not WC_HASH_TYPE_NONE and does not match the hash type used during initialization.
- HASH_TYPE_E the hash type stored in the hash object is not supported or not compiled in.
- Other negative error codes may be returned by the underlying hash algorithm's update function.

Example

```
wc_HashAlg hash;
byte digest[WC_SHA256_DIGEST_SIZE];
byte data[] = { 0x61, 0x62, 0x63 };
int ret;

ret = wc_HashInit(&hash, WC_HASH_TYPE_SHA256);
if (ret == 0) {
    ret = wc_HashUpdate(&hash, WC_HASH_TYPE_SHA256, data, sizeof(data));
    if (ret == 0) {
        ret = wc_HashFinal(&hash, WC_HASH_TYPE_SHA256, digest);
    }
    wc_HashFree(&hash, WC_HASH_TYPE_SHA256);
}
```

```
int wc_HashFinal(
    wc_HashAlg * hash,
    enum wc_HashType type,
    byte * out
)
```

This function finalizes the hash in a wc_HashAlg object and writes the resulting digest to out. The hash algorithm used is the one stored in the wc_HashAlg object when it was initialized with wc_HashInit() or wc_HashInit_ex().

Parameters:

- **hash** Pointer to the wc_HashAlg object, previously initialized with wc_HashInit() or wc_HashInit_ex().
- **type** Must be either WC_HASH_TYPE_NONE or the same hash type that was used to initialize the hash object. The hash type stored in the hash object during initialization is always used to select the algorithm. Any other value results in BAD_FUNC_ARG.
- **out** Pointer to the buffer to receive the digest. It must be at least as large as the digest size of the hash type the object was initialized with (see wc_HashGetDigestSize()).

See:

- wc_HashUpdate
- wc_HashFree
- wc_HashGetDigestSize

Return:

- 0 Success.
- BAD_FUNC_ARG hash or out is NULL, the hash object was not initialized with a valid hash type, or type is not WC_HASH_TYPE_NONE and does not match the hash type used during initialization.
- HASH_TYPE_E the hash type stored in the hash object is not supported or not compiled in.
- Other negative error codes may be returned by the underlying hash algorithm's final function.

Example

```
wc_HashAlg hash;
byte digest[WC_SHA256_DIGEST_SIZE];
byte data[] = { 0x61, 0x62, 0x63 };
int ret;

ret = wc_HashInit(&hash, WC_HASH_TYPE_SHA256);
if (ret == 0) {
    ret = wc_HashUpdate(&hash, WC_HASH_TYPE_SHA256, data, sizeof(data));
    if (ret == 0) {
        ret = wc_HashFinal(&hash, WC_HASH_TYPE_SHA256, digest);
    }
    wc_HashFree(&hash, WC_HASH_TYPE_SHA256);
}
```

```
int wc_HashFree(
    wc_HashAlg * hash,
    enum wc_HashType type
)
```

This function frees any resources held by a wc_HashAlg object. The hash algorithm used is the one stored in the wc_HashAlg object when it was initialized with wc_HashInit() or wc_HashInit_ex(). The wc_HashAlg object itself is not freed.

Parameters:

- **hash** Pointer to the wc_HashAlg object, previously initialized with wc_HashInit() or wc_HashInit_ex().
- **type** Must be either WC_HASH_TYPE_NONE or the same hash type that was used to initialize the hash object. The hash type stored in the hash object during initialization is always used to select the algorithm. Any other value results in BAD_FUNC_ARG.

See:

- wc_HashUpdate

- wc_HashFinal

Return:

- 0 Success.
- BAD_FUNC_ARG hash is NULL, the hash object was not initialized with a valid hash type, or type is not WC_HASH_TYPE_NONE and does not match the hash type used during initialization.
- HASH_TYPE_E the hash type stored in the hash object is not supported or not compiled in.

Example

```
wc_HashAlg hash;

if (wc_HashInit(&hash, WC_HASH_TYPE_SHA256) == 0) {
    // use hash
    wc_HashFree(&hash, WC_HASH_TYPE_SHA256);
}
```

```
int wc_HashSetFlags(
    wc_HashAlg * hash,
    enum wc_HashType type,
    word32 flags
)
```

This function sets the flags of the underlying hash object in a wc_HashAlg object. The hash algorithm used is the one stored in the wc_HashAlg object when it was initialized with wc_HashInit() or wc_HashInit_ex(). Available when WOLFSSL_HASH_FLAGS is defined.

Parameters:

- **hash** Pointer to the wc_HashAlg object, previously initialized with wc_HashInit() or wc_HashInit_ex().
- **type** Must be either WC_HASH_TYPE_NONE or the same hash type that was used to initialize the hash object. The hash type stored in the hash object during initialization is always used to select the algorithm. Any other value results in BAD_FUNC_ARG.
- **flags** Flags to set (for example WC_HASH_FLAG_ISCOPY).

See: wc_HashGetFlags

Return:

- 0 Success.
- BAD_FUNC_ARG hash is NULL, the hash object was not initialized with a valid hash type, or type is not WC_HASH_TYPE_NONE and does not match the hash type used during initialization.
- HASH_TYPE_E the hash type stored in the hash object is not supported or not compiled in.

Example

```
wc_HashAlg hash;

if (wc_HashInit(&hash, WC_HASH_TYPE_SHA256) == 0) {
```

```
    wc_HashSetFlags(&hash, WC_HASH_TYPE_SHA256, WC_HASH_FLAG_NONE);
    wc_HashFree(&hash, WC_HASH_TYPE_SHA256);
}
```

```
int wc_HashGetFlags(
    wc_HashAlg * hash,
    enum wc_HashType type,
    word32 * flags
)
```

This function gets the flags of the underlying hash object in a wc_HashAlg object. The hash algorithm used is the one stored in the wc_HashAlg object when it was initialized with wc_HashInit() or wc_HashInit_ex(). Available when WOLFSSL_HASH_FLAGS is defined.

Parameters:

- **hash** Pointer to the wc_HashAlg object, previously initialized with wc_HashInit() or wc_HashInit_ex().
- **type** Must be either WC_HASH_TYPE_NONE or the same hash type that was used to initialize the hash object. The hash type stored in the hash object during initialization is always used to select the algorithm. Any other value results in BAD_FUNC_ARG.
- **flags** Pointer to receive the current flags. If NULL, no flags are returned.

See: wc_HashSetFlags

Return:

- 0 Success.
- BAD_FUNC_ARG hash is NULL, the hash object was not initialized with a valid hash type, or type is not WC_HASH_TYPE_NONE and does not match the hash type used during initialization.
- HASH_TYPE_E the hash type stored in the hash object is not supported or not compiled in.

Example

```
wc_HashAlg hash;
word32 flags = 0;

if (wc_HashInit(&hash, WC_HASH_TYPE_SHA256) == 0) {
    wc_HashGetFlags(&hash, WC_HASH_TYPE_SHA256, &flags);
    wc_HashFree(&hash, WC_HASH_TYPE_SHA256);
}
```

```
int wc_UnpackOctets(
    byte * out,
    word32 outSz,
    const byte * in,
    word32 inSz,
    word32 octetSz
)
```

Expands a packed octet stream into the one-octet-per-byte-cell form every wolfCrypt byte* API expects. Declared only when WOLFSSL_WIDE_BYTE is set (CHAR_BIT != 8); elsewhere the two layouts coincide and no conversion is needed.

Parameters:

- **out** Destination, one octet per cell
- **outSz** Size of out in byte cells
- **in** Packed source, WC_OCTETS_PER_BYTE octets per cell, low octet first, and must not overlap out
- **inSz** Size of in in byte cells; must be at least WC_PACKED_CELLS(octetSz)
- **octetSz** Number of octets to expand

See: wc_PackOctets

Return:

- 0 on success
- BAD_FUNC_ARG when out or in is NULL
- BUFFER_E when out is smaller than octetSz, or in smaller than WC_PACKED_CELLS(octetSz)

Example

```
byte sig[WC_MLDSA_65_SIG_SIZE];
int ret = wc_UnpackOctets(sig, (word32)sizeof(sig), sigPacked,
                          (word32)sizeof(sigPacked),
                          WC_MLDSA_65_SIG_SIZE);
```

```
int wc_PackOctets(
    byte * out,
    word32 outSz,
    const byte * in,
    word32 inSz,
    word32 octetSz
)
```

Packs a one-octet-per-byte-cell buffer into an octet stream, for storing to flash or handing to a byte-oriented peripheral. Inverse of wc_UnpackOctets(); requires WOLFSSL_WIDE_BYTE.

Parameters:

- **out** Destination for the packed stream
- **outSz** Size of out in byte cells
- **in** Source, one octet per cell, and must not overlap out
- **inSz** Size of in in byte cells; must be at least octetSz
- **octetSz** Number of octets to pack

See: wc_UnpackOctets

Return:

- 0 on success

- BAD_FUNC_ARG when out or in is NULL
- BUFFER_E when out is smaller than WC_PACKED_CELLS(octetSz), or in smaller than octetSz

Example

```
byte packed[WC_PACKED_CELLS(sizeof(sig))];
int ret = wc_PackOctets(packed, (word32)sizeof(packed), sig,
                        (word32)sizeof(sig), (word32)sizeof(sig));
```

```
int wolfCrypt_Init(
    void
)
```

Used to initialize resources used by wolfCrypt.

Parameters:

- **none** No parameters.

See: wolfCrypt_Cleanup

Return:

- 0 upon success.
- <0 upon failure of init resources.

Example

```
...
if (wolfCrypt_Init() != 0) {
    WOLFSSL_MSG("Error with wolfCrypt_Init call");
}
```

```
int wolfCrypt_Cleanup(
    void
)
```

Used to clean up resources used by wolfCrypt.

Parameters:

- **none** No parameters.

See: wolfCrypt_Init

Return:

- 0 upon success.

- <0 upon failure of cleaning up resources.

Example

```
...
if (wolfCrypt_Cleanup() != 0) {
    WOLFSSL_MSG("Error with wolfCrypt_Cleanup call");
}
```

B.11 Algorithms - 3DES

B.10.2.12 function wolfCrypt_Cleanup

B.11.1 Functions

	Name
int	**wc_Des_SetKey**(Des * des, const byte * key, const byte * iv, int dir)This function sets the key and initialization vector (iv) for the Des structure given as argument. It also initializes and allocates space for the buffers needed for encryption and decryption, if these have not yet been initialized. Note: If no iv is provided (i.e. iv == NULL) the initialization vector defaults to an iv of 0.
void	**wc_Des_SetIV**(Des * des, const byte * iv)This function sets the initialization vector (iv) for the Des structure given as argument. When passed a NULL iv, it sets the initialization vector to 0.
int	**wc_Des_CbcEncrypt**(Des * des, byte * out, const byte * in, word32 sz)This function encrypts the input message, in, and stores the result in the output buffer, out. It uses DES encryption with cipher block chaining (CBC) mode.
int	**wc_Des_CbcDecrypt**(Des * des, byte * out, const byte * in, word32 sz)This function decrypts the input ciphertext, in, and stores the resulting plaintext in the output buffer, out. It uses DES encryption with cipher block chaining (CBC) mode.
int	**wc_Des_EcbEncrypt**(Des * des, byte * out, const byte * in, word32 sz)This function encrypts the input message, in, and stores the result in the output buffer, out. It uses Des encryption with Electronic Codebook (ECB) mode.

	Name
int	**wc_Des3_EcbEncrypt**(Des3 * des, byte * out, const byte * in, word32 sz)This function encrypts the input message, in, and stores the result in the output buffer, out. It uses Des3 encryption with Electronic Codebook (ECB) mode. Warning: In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB API's directly whenever possible.
int	**wc_Des3_SetKey**(Des3 * des, const byte * key, const byte * iv, int dir)This function sets the key and initialization vector (iv) for the Des3 structure given as argument. It also initializes and allocates space for the buffers needed for encryption and decryption, if these have not yet been initialized. Note: If no iv is provided (i.e. iv == NULL) the initialization vector defaults to an iv of 0.
int	**wc_Des3_SetIV**(Des3 * des, const byte * iv)This function sets the initialization vector (iv) for the Des3 structure given as argument. When passed a NULL iv, it sets the initialization vector to 0.
int	**wc_Des3_CbcEncrypt**(Des3 * des, byte * out, const byte * in, word32 sz)This function encrypts the input message, in, and stores the result in the output buffer, out. It uses Triple Des (3DES) encryption with cipher block chaining (CBC) mode.
int	**wc_Des3_CbcDecrypt**(Des3 * des, byte * out, const byte * in, word32 sz)This function decrypts the input ciphertext, in, and stores the resulting plaintext in the output buffer, out. It uses Triple Des (3DES) encryption with cipher block chaining (CBC) mode.
int	**wc_Des_EcbDecrypt**(Des * des, byte * out, const byte * in, word32 sz)This function decrypts the input ciphertext and stores the resulting plaintext in the output buffer. It uses DES encryption with Electronic Codebook (ECB) mode. Warning: In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB APIs directly whenever possible.

	Name
int	**wc_Des3_EcbDecrypt**(Des3 * des, byte * out, const byte * in, word32 sz)This function decrypts the input ciphertext and stores the resulting plaintext in the output buffer. It uses Triple DES (3DES) encryption with Electronic Codebook (ECB) mode. Warning: In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB APIs directly whenever possible.
int	**wc_Des3Init**(Des3 * des3, void * heap, int devId)This function initializes a Des3 structure for use with hardware acceleration and custom memory management. This is an extended version of the standard initialization that allows specification of heap hints and device IDs.
void	**wc_Des3Free**(Des3 * des3)This function frees a Des3 structure and releases any resources allocated for it. This should be called when finished using the Des3 structure to prevent memory leaks.
int	**wc_Des_CbcDecryptWithKey**(byte * out, const byte * in, word32 sz, const byte * key, const byte * iv)This function decrypts the input ciphertext, in, and stores the resulting plaintext in the output buffer, out. It uses DES encryption with cipher block chaining (CBC) mode. This function is a substitute for wc_Des_CbcDecrypt, allowing the user to decrypt a message without directly instantiating a Des structure.
int	**wc_Des_CbcEncryptWithKey**(byte * out, const byte * in, word32 sz, const byte * key, const byte * iv)This function encrypts the input plaintext, in, and stores the resulting ciphertext in the output buffer, out. It uses DES encryption with cipher block chaining (CBC) mode. This function is a substitute for wc_Des_CbcEncrypt, allowing the user to encrypt a message without directly instantiating a Des structure.
int	**wc_Des3_CbcEncryptWithKey**(byte * out, const byte * in, word32 sz, const byte * key, const byte * iv)This function encrypts the input plaintext, in, and stores the resulting ciphertext in the output buffer, out. It uses Triple DES (3DES) encryption with cipher block chaining (CBC) mode. This function is a substitute for wc_Des3_CbcEncrypt, allowing the user to encrypt a message without directly instantiating a Des3 structure.

	Name
int	**wc_Des3_CbcDecryptWithKey**(byte * out, const byte * in, word32 sz, const byte * key, const byte * iv)This function decrypts the input ciphertext, in, and stores the resulting plaintext in the output buffer, out. It uses Triple Des (3DES) encryption with cipher block chaining (CBC) mode. This function is a substitute for wc_Des3_CbcDecrypt, allowing the user to decrypt a message without directly instantiating a Des3 structure.

B.11.2 Functions Documentation

```
int wc_Des_SetKey(
    Des * des,
    const byte * key,
    const byte * iv,
    int dir
)
```

This function sets the key and initialization vector (iv) for the Des structure given as argument. It also initializes and allocates space for the buffers needed for encryption and decryption, if these have not yet been initialized. Note: If no iv is provided (i.e. iv == NULL) the initialization vector defaults to an iv of 0.

Parameters:

- **des** pointer to the Des structure to initialize
- **key** pointer to the buffer containing the 8 byte key with which to initialize the Des structure
- **iv** pointer to the buffer containing the 8 byte iv with which to initialize the Des structure. If this is not provided, the iv defaults to 0
- **dir** direction of encryption. Valid options are: DES_ENCRYPTION, and DES_DECRYPTION

See:

- wc_Des_SetIV
- wc_Des3_SetKey

Return: 0 On successfully setting the key and initialization vector for the Des structure

Note: This function must be called on a Des structure before it is passed to wc_Des_CbcEncrypt(), wc_Des_CbcDecrypt(), wc_Des_EcbEncrypt(), or wc_Des_EcbDecrypt(). Unlike the Des3 API, the Des structure has no key-set state, so those functions do not verify that a key has been set and will operate on undefined key material if this precondition is not met.

3

Example

```
Des enc; // Des structure used for encryption
int ret;
byte key[] = { // initialize with 8 byte key };
```

```
byte iv[]  = { // initialize with 8 byte iv };

ret = wc_Des_SetKey(&des, key, iv, DES_ENCRYPTION);
if (ret != 0) {
    // error initializing des structure
}
```

```
void wc_Des_SetIV(
    Des * des,
    const byte * iv
)
```

This function sets the initialization vector (iv) for the Des structure given as argument. When passed a NULL iv, it sets the initialization vector to 0.

Parameters:

- **des** pointer to the Des structure for which to set the iv
- **iv** pointer to the buffer containing the 8 byte iv with which to initialize the Des structure. If this is not provided, the iv defaults to 0

See: wc_Des_SetKey

Return: none No returns.

3

Example

```
Des enc; // Des structure used for encryption
// initialize enc with wc_Des_SetKey
byte iv[]  = { // initialize with 8 byte iv };
wc_Des_SetIV(&enc, iv);
}
```

```
int wc_Des_CbcEncrypt(
    Des * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function encrypts the input message, in, and stores the result in the output buffer, out. It uses DES encryption with cipher block chaining (CBC) mode.

Parameters:

- **des** pointer to the Des structure to use for encryption
- **out** pointer to the buffer in which to store the encrypted ciphertext
- **in** pointer to the input buffer containing the message to encrypt

- **sz** length of the message to encrypt

See:

- wc_Des_SetKey
- wc_Des_CbcDecrypt

Return: 0 Returned upon successfully encrypting the given input message

Note: wc_Des_SetKey() must be called on des before calling this function. This function does not check whether a key has been set and will encrypt using undefined key material if it has not.

3

Example

```
Des enc; // Des structure used for encryption
// initialize enc with wc_Des_SetKey, use mode DES_ENCRYPTION

byte plain[]  = { // initialize with message };
byte cipher[sizeof(plain)];

if ( wc_Des_CbcEncrypt(&enc, cipher, plain, sizeof(plain)) != 0) {
    // error encrypting message
}
```

```
int wc_Des_CbcDecrypt(
    Des * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function decrypts the input ciphertext, in, and stores the resulting plaintext in the output buffer, out. It uses DES encryption with cipher block chaining (CBC) mode.

Parameters:

- **des** pointer to the Des structure to use for decryption
- **out** pointer to the buffer in which to store the decrypted plaintext
- **in** pointer to the input buffer containing the encrypted ciphertext
- **sz** length of the ciphertext to decrypt

See:

- wc_Des_SetKey
- wc_Des_CbcEncrypt

Return: 0 Returned upon successfully decrypting the given ciphertext

Note: wc_Des_SetKey() must be called on des before calling this function. This function does not check whether a key has been set and will decrypt using undefined key material if it has not.

3

Example

```
Des dec; // Des structure used for decryption
// initialize dec with wc_Des_SetKey, use mode DES_DECRYPTION

byte cipher[]  = { // initialize with ciphertext };
byte decoded[sizeof(cipher)];

if ( wc_Des_CbcDecrypt(&dec, decoded, cipher, sizeof(cipher)) != 0) {
    // error decrypting message
}
```

```
int wc_Des_EcbEncrypt(
    Des * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function encrypts the input message, in, and stores the result in the output buffer, out. It uses Des encryption with Electronic Codebook (ECB) mode.

Parameters:

- **des** pointer to the Des structure to use for encryption
- **out** pointer to the buffer in which to store the encrypted message
- **in** pointer to the input buffer containing the plaintext to encrypt
- **sz** length of the plaintext to encrypt

See: wc_Des_SetKey

Return: 0: Returned upon successfully encrypting the given plaintext.

Note: wc_Des_SetKey() must be called on des before calling this function. This function does not check whether a key has been set and will encrypt using undefined key material if it has not.

3

Example

```
Des enc; // Des structure used for encryption
// initialize enc with wc_Des_SetKey, use mode DES_ENCRYPTION

byte plain[]  = { // initialize with message to encrypt };
byte cipher[sizeof(plain)];

if ( wc_Des_EcbEncrypt(&enc,cipher, plain, sizeof(plain)) != 0) {
    // error encrypting message
}
```

```
int wc_Des3_EcbEncrypt(
    Des3 * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function encrypts the input message, in, and stores the result in the output buffer, out. It uses Des3 encryption with Electronic Codebook (ECB) mode. Warning: In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB API's directly whenever possible.

Parameters:

- **des** pointer to the Des3 structure to use for encryption
- **out** pointer to the buffer in which to store the encrypted message
- **in** pointer to the input buffer containing the plaintext to encrypt
- **sz** length of the plaintext to encrypt

See: wc_Des3_SetKey

Return:

- 0 Returned upon successfully encrypting the given plaintext
- BAD_FUNC_ARG If des, out, or in is NULL.
- MISSING_KEY If no key has been set on the Des3 structure with wc_Des3_SetKey.

3

Example

```
Des3 enc; // Des3 structure used for encryption
// initialize enc with wc_Des3_SetKey, use mode DES_ENCRYPTION

byte plain[]  = { // initialize with message to encrypt };
byte cipher[sizeof(plain)];

if ( wc_Des3_EcbEncrypt(&enc,cipher, plain, sizeof(plain)) != 0) {
    // error encrypting message
}
```

```
int wc_Des3_SetKey(
    Des3 * des,
    const byte * key,
    const byte * iv,
    int dir
)
```

This function sets the key and initialization vector (iv) for the Des3 structure given as argument. It also initializes and allocates space for the buffers needed for encryption and decryption, if these have not yet been initialized. Note: If no iv is provided (i.e. iv == NULL) the initialization vector defaults to an iv of 0.

Parameters:

- **des3** pointer to the Des3 structure to initialize
- **key** pointer to the buffer containing the 24 byte key with which to initialize the Des3 structure
- **iv** pointer to the buffer containing the 8 byte iv with which to initialize the Des3 structure. If this is not provided, the iv defaults to 0
- **dir** direction of encryption. Valid options are: DES_ENCRYPTION, and DES_DECRYPTION

See:

- wc_Des3_SetIV
- wc_Des3_CbcEncrypt
- wc_Des3_CbcDecrypt

Return: 0 On successfully setting the key and initialization vector for the Des structure

3

Example

```
Des3 enc; // Des3 structure used for encryption
int ret;
byte key[] = { // initialize with 24 byte key };
byte iv[]  = { // initialize with 8 byte iv };

ret = wc_Des3_SetKey(&des, key, iv, DES_ENCRYPTION);
if (ret != 0) {
    // error initializing des structure
}
```

```
int wc_Des3_SetIV(
    Des3 * des,
    const byte * iv
)
```

This function sets the initialization vector (iv) for the Des3 structure given as argument. When passed a NULL iv, it sets the initialization vector to 0.

Parameters:

- **des** pointer to the Des3 structure for which to set the iv
- **iv** pointer to the buffer containing the 8 byte iv with which to initialize the Des3 structure. If this is not provided, the iv defaults to 0

See: wc_Des3_SetKey

Return: none No returns.

3

Example

```
Des3 enc; // Des3 structure used for encryption
// initialize enc with wc_Des3_SetKey

byte iv[]  = { // initialize with 8 byte iv };

wc_Des3_SetIV(&enc, iv);
}



int wc_Des3_CbcEncrypt(
    Des3 * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function encrypts the input message, in, and stores the result in the output buffer, out. It uses Triple Des (3DES) encryption with cipher block chaining (CBC) mode.

Parameters:

- **des** pointer to the Des3 structure to use for encryption
- **out** pointer to the buffer in which to store the encrypted ciphertext
- **in** pointer to the input buffer containing the message to encrypt
- **sz** length of the message to encrypt

See:

- wc_Des3_SetKey
- wc_Des3_CbcDecrypt

Return:

- 0 Returned upon successfully encrypting the given input message
- BAD_FUNC_ARG If des, out, or in is NULL.
- BAD_LENGTH_E If sz is not a multiple of DES_BLOCK_SIZE.
- MISSING_KEY If no key has been set on the Des3 structure with wc_Des3_SetKey.

3

Example

```
Des3 enc; // Des3 structure used for encryption
// initialize enc with wc_Des3_SetKey, use mode DES_ENCRYPTION

byte plain[]  = { // initialize with message };
byte cipher[sizeof(plain)];

if ( wc_Des3_CbcEncrypt(&enc, cipher, plain, sizeof(plain)) != 0) {
    // error encrypting message
}
```

```
int wc_Des3_CbcDecrypt(
    Des3 * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function decrypts the input ciphertext, in, and stores the resulting plaintext in the output buffer, out. It uses Triple Des (3DES) encryption with cipher block chaining (CBC) mode.

Parameters:

- **des** pointer to the Des3 structure to use for decryption
- **out** pointer to the buffer in which to store the decrypted plaintext
- **in** pointer to the input buffer containing the encrypted ciphertext
- **sz** length of the ciphertext to decrypt

See:

- wc_Des3_SetKey
- wc_Des3_CbcEncrypt

Return:

- 0 Returned upon successfully decrypting the given ciphertext
- BAD_FUNC_ARG If des, out, or in is NULL.
- BAD_LENGTH_E If sz is not a multiple of DES_BLOCK_SIZE.
- MISSING_KEY If no key has been set on the Des3 structure with wc_Des3_SetKey.

3

Example

```
Des3 dec; // Des structure used for decryption
// initialize dec with wc_Des3_SetKey, use mode DES_DECRYPTION

byte cipher[]  = { // initialize with ciphertext };
byte decoded[sizeof(cipher)];

if ( wc_Des3_CbcDecrypt(&dec, decoded, cipher, sizeof(cipher)) != 0) {
    // error decrypting message
}
```

```
int wc_Des_EcbDecrypt(
    Des * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function decrypts the input ciphertext and stores the resulting plaintext in the output buffer. It uses DES encryption with Electronic Codebook (ECB) mode. Warning: In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB APIs directly whenever possible.

Parameters:

- **des** pointer to the Des structure to use for decryption
- **out** pointer to the buffer in which to store the decrypted plaintext
- **in** pointer to the input buffer containing the ciphertext
- **sz** length of the ciphertext to decrypt

See:

- wc_Des_SetKey
- wc_Des_EcbEncrypt

Return: 0 On successfully decrypting the given ciphertext

Note: wc_Des_SetKey() must be called on des before calling this function. This function does not check whether a key has been set and will decrypt using undefined key material if it has not.

3

Example

```
Des dec;
byte cipher[]; // ciphertext to decrypt
byte plain[sizeof(cipher)];

wc_Des_SetKey(&dec, key, iv, DES_DECRYPTION);
if (wc_Des_EcbDecrypt(&dec, plain, cipher, sizeof(cipher)) != 0) {
    // error decrypting message
}
```

```
int wc_Des3_EcbDecrypt(
    Des3 * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function decrypts the input ciphertext and stores the resulting plaintext in the output buffer. It uses Triple DES (3DES) encryption with Electronic Codebook (ECB) mode. Warning: In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB APIs directly whenever possible.

Parameters:

- **des** pointer to the Des3 structure to use for decryption
- **out** pointer to the buffer in which to store the decrypted plaintext
- **in** pointer to the input buffer containing the ciphertext
- **sz** length of the ciphertext to decrypt

See:

- wc_Des3_SetKey
- wc_Des3_EcbEncrypt

Return:

- 0 On successfully decrypting the given ciphertext
- BAD_FUNC_ARG If des, out, or in is NULL.
- MISSING_KEY If no key has been set on the Des3 structure with wc_Des3_SetKey.

3

Example

```
Des3 dec;
byte cipher[]; // ciphertext to decrypt
byte plain[sizeof(cipher)];

wc_Des3_SetKey(&dec, key, iv, DES_DECRYPTION);
if (wc_Des3_EcbDecrypt(&dec, plain, cipher, sizeof(cipher)) != 0) {
    // error decrypting message
}
```

```
int wc_Des3Init(
    Des3 * des3,
    void * heap,
    int devId
)
```

This function initializes a Des3 structure for use with hardware acceleration and custom memory management. This is an extended version of the standard initialization that allows specification of heap hints and device IDs.

Parameters:

- **des3** pointer to the Des3 structure to initialize
- **heap** pointer to heap hint for memory allocation (can be NULL)
- **devId** device ID for hardware acceleration (use INVALID_DEVID for software only)

See:

- wc_Des3_SetKey
- wc_Des3Free

Return:

- 0 On successfully initializing the Des3 structure
- BAD_FUNC_ARG If des3 is NULL

3

Example

```
Des3 des;
void* heap = NULL;
int devId = INVALID_DEVID;

if (wc_Des3Init(&des, heap, devId) != 0) {
    // error initializing Des3 structure
}
```

```
void wc_Des3Free(
    Des3 * des3
)
```

This function frees a Des3 structure and releases any resources allocated for it. This should be called when finished using the Des3 structure to prevent memory leaks.

Parameters:

- **des3** pointer to the Des3 structure to free

See:

- wc_Des3Init
- wc_Des3_SetKey

Return: none No returns.

3

Example

```
Des3 des;
wc_Des3Init(&des, NULL, INVALID_DEVID);
wc_Des3_SetKey(&des, key, iv, DES_ENCRYPTION);
// use des for encryption/decryption
wc_Des3Free(&des);
```

```
int wc_Des_CbcDecryptWithKey(
    byte * out,
    const byte * in,
    word32 sz,
    const byte * key,
    const byte * iv
)
```

This function decrypts the input ciphertext, in, and stores the resulting plaintext in the output buffer, out. It uses DES encryption with cipher block chaining (CBC) mode. This function is a substitute for wc_Des_CbcDecrypt, allowing the user to decrypt a message without directly instantiating a Des structure.

Parameters:

- **out** pointer to the buffer in which to store the decrypted plaintext
- **in** pointer to the input buffer containing the encrypted ciphertext
- **sz** length of the ciphertext to decrypt
- **key** pointer to the buffer containing the 8 byte key to use for decryption
- **iv** pointer to the buffer containing the 8 byte iv to use for decryption. If no iv is provided, the iv defaults to 0

See: wc_Des_CbcDecrypt

Return:

- 0 Returned upon successfully decrypting the given ciphertext
- MEMORY_E Returned if there is an error allocating space for a Des structure

3

Example

```
int ret;
byte key[] = { // initialize with 8 byte key };
byte iv[]  = { // initialize with 8 byte iv };

byte cipher[]  = { // initialize with ciphertext };
byte decoded[sizeof(cipher)];

if ( wc_Des_CbcDecryptWithKey(decoded, cipher, sizeof(cipher), key,
iv) != 0) {
    // error decrypting message
}
```

```
int wc_Des_CbcEncryptWithKey(
    byte * out,
    const byte * in,
    word32 sz,
    const byte * key,
    const byte * iv
)
```

This function encrypts the input plaintext, in, and stores the resulting ciphertext in the output buffer, out. It uses DES encryption with cipher block chaining (CBC) mode. This function is a substitute for wc_Des_CbcEncrypt, allowing the user to encrypt a message without directly instantiating a Des structure.

Parameters:

- **out** Final encrypted data
- **in** Data to be encrypted, must be padded to Des block size.
- **sz** Size of input buffer.
- **key** Pointer to the key to use for encryption.
- **iv** Initialization vector

See:

- wc_Des_CbcDecryptWithKey
- wc_Des_CbcEncrypt

Return:

- 0 Returned after successfully encrypting data.
- MEMORY_E Returned if there's an error allocating memory for a Des structure.
- <0 Returned on any error during encryption.

3

Example

```
byte key[] = { // initialize with 8 byte key };
byte iv[]  = { // initialize with 8 byte iv };
byte in[] = { // Initialize with plaintext };
byte out[sizeof(in)];
if ( wc_Des_CbcEncryptWithKey(&out, in, sizeof(in), key, iv) != 0)
{
    // error encrypting message
}
```

```
int wc_Des3_CbcEncryptWithKey(
    byte * out,
    const byte * in,
    word32 sz,
    const byte * key,
    const byte * iv
)
```

This function encrypts the input plaintext, in, and stores the resulting ciphertext in the output buffer, out. It uses Triple DES (3DES) encryption with cipher block chaining (CBC) mode. This function is a substitute for wc_Des3_CbcEncrypt, allowing the user to encrypt a message without directly instantiating a Des3 structure.

Parameters:

- **out** Final encrypted data
- **in** Data to be encrypted, must be padded to Des block size.
- **sz** Size of input buffer.
- **key** Pointer to the key to use for encryption.
- **iv** Initialization vector

See:

- wc_Des3_CbcDecryptWithKey
- wc_Des_CbcEncryptWithKey
- wc_Des_CbcDecryptWithKey

Return:

- 0 Returned after successfully encrypting data.
- MEMORY_E Returned if there's an error allocating memory for a Des structure.
- <0 Returned on any error during encryption.

3

Example

```
byte key[] = { // initialize with 8 byte key };
byte iv[]  = { // initialize with 8 byte iv };

byte in[] = { // Initialize with plaintext };
byte out[sizeof(in)];

if ( wc_Des3_CbcEncryptWithKey(&out, in, sizeof(in), key, iv) != 0)
{
    // error encrypting message
}
```

```
int wc_Des3_CbcDecryptWithKey(
    byte * out,
    const byte * in,
    word32 sz,
    const byte * key,
    const byte * iv
)
```

This function decrypts the input ciphertext, in, and stores the resulting plaintext in the output buffer, out. It uses Triple Des (3DES) encryption with cipher block chaining (CBC) mode. This function is a substitute for wc_Des3_CbcDecrypt, allowing the user to decrypt a message without directly instantiating a Des3 structure.

Parameters:

- **out** pointer to the buffer in which to store the decrypted plaintext
- **in** pointer to the input buffer containing the encrypted ciphertext
- **sz** length of the ciphertext to decrypt
- **key** pointer to the buffer containing the 24 byte key to use for decryption
- **iv** pointer to the buffer containing the 8 byte iv to use for decryption. If no iv is provided, the iv defaults to 0

See: wc_Des3_CbcDecrypt

Return:

- 0 Returned upon successfully decrypting the given ciphertext
- MEMORY_E Returned if there is an error allocating space for a Des structure

3

Example

```
int ret;
byte key[] = { // initialize with 24 byte key };
byte iv[]  = { // initialize with 8 byte iv };

byte cipher[]  = { // initialize with ciphertext };
byte decoded[sizeof(cipher)];

if ( wc_Des3_CbcDecryptWithKey(decoded, cipher, sizeof(cipher),
key, iv) != 0) {
    // error decrypting message
}
```

B.12 Algorithms - AES

B.11.2.18 function wc_Des3_CbcDecryptWithKey

B.12.1 Functions

	Name
int	**wc_AesSetKey**(Aes * aes, const byte * key, word32 len, const byte * iv, int dir)This function initializes an AES structure by setting the key and then setting the initialization vector.
int	**wc_AesSetIV**(Aes * aes, const byte * iv)This function sets the initialization vector for a particular AES object. The AES object should be initialized before calling this function.

	Name
int	**wc_AesCbcEncrypt**(Aes * aes, byte * out, const byte * in, word32 sz)Encrypts a plaintext message from the input buffer in, and places the resulting cipher text in the output buffer out using cipher block chaining with AES. This function requires that the AES object has been initialized by calling AesSetKey before a message is able to be encrypted. This function assumes that the input message is AES block length aligned, and expects the input length to be a multiple of the block length, which will optionally be checked and enforced if WOLFSSL_AES_CBC_LENGTH_CHECKS is defined in the build configuration. In order to assure block-multiple input, PKCS#7 style padding should be added beforehand. This differs from the OpenSSL AES-CBC methods which add the padding for you. To make the wolfSSL and corresponding OpenSSL functions interoperate, one should specify the -nopad option in the OpenSSL command line function so that it behaves like the wolfSSL AesCbcEncrypt method and does not add extra padding during encryption.
int	**wc_AesCbcDecrypt**(Aes * aes, byte * out, const byte * in, word32 sz)Decrypts a cipher from the input buffer in, and places the resulting plain text in the output buffer out using cipher block chaining with AES. This function requires that the AES structure has been initialized by calling AesSetKey before a message is able to be decrypted. This function assumes that the original message was AES block length aligned, and expects the input length to be a multiple of the block length, which will optionally be checked and enforced if WOLFSSL_AES_CBC_LENGTH_CHECKS is defined in the build configuration. This differs from the OpenSSL AES-CBC methods, which add PKCS#7 padding automatically, and so do not require block-multiple input. To make the wolfSSL function and equivalent OpenSSL functions interoperate, one should specify the -nopad option in the OpenSSL command line function so that it behaves like the wolfSSL AesCbcEncrypt method and does not create errors during decryption.

	Name
int	**wc_AesCtrEncrypt**(Aes * aes, byte * out, const byte * in, word32 sz)Encrypts/Decrypts a message from the input buffer in, and places the resulting cipher text in the output buffer out using CTR mode with AES. This function is only enabled if WOLFSSL_AES_COUNTER is enabled at compile time. The AES structure should be initialized through AesSetKey before calling this function. Note that this function is used for both decryption and encryption. *NOTE:* Regarding using same API for encryption and decryption. User should differentiate between Aes structures for encrypt/decrypt.
int	**wc_AesEncryptDirect**(Aes * aes, byte * out, const byte * in)This function is a one_block encrypt of the input block, in, into the output block, out. It uses the key of the provided AES structure, which should be initialized with wc_AesSetKey before calling this function. wc_AesSetKey should have been called with the iv set to NULL. This is only enabled if the configure option WOLFSSL_AES_DIRECT is enabled. **Warning:** In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB API's directly whenever possible. When WC_AES_DIRECT_CRYPTO_CB is defined (set automatically for WOLF_CRYPTO_CB builds that use the generic AES Direct code, not a port's own version), if the AES object has a crypto callback device (devId), that device is asked first to encrypt the block as a one-block ECB request. The software key is used only if the device declines, and a device error is returned as-is. A device that owns the key must handle the request or return an error; it must not return CRYPTOCB_UNAVAILABLE. With WOLF_CRYPTO_CB_FIND, the device find callback picks the device, so a device can be asked even when the AES object has no devId.

	Name
int	**wc_AesDecryptDirect**(Aes * aes, byte * out, const byte * in)This function is a one_block decrypt of the input block, in, into the output block, out. It uses the key of the provided AES structure, which should be initialized with wc_AesSetKey before calling this function. wc_AesSetKey should have been called with the iv set to NULL. This is only enabled if the configure option WOLFSSL_AES_DIRECT is enabled. **Warning:** In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB API's directly whenever possible. When WC_AES_DIRECT_CRYPTO_CB is defined (set automatically for WOLF_CRYPTO_CB builds that use the generic AES Direct code, not a port's own version), if the AES object has a crypto callback device (devId), that device is asked first to decrypt the block as a one-block ECB request. The software key is used only if the device declines, and a device error is returned as-is. A device that owns the key must handle the request or return an error; it must not return CRYPTOCB_UNAVAILABLE. With WOLF_CRYPTO_CB_FIND, the device find callback picks the device, so a device can be asked even when the AES object has no devId.
int	**wc_AesSetKeyDirect**(Aes * aes, const byte * key, word32 len, const byte * iv, int dir)This function is used to set the AES keys for CTR mode with AES. It initializes an AES object with the given key, iv (initialization vector), and encryption dir (direction). It is only enabled if the configure option WOLFSSL_AES_DIRECT is enabled. Currently wc_AesSetKeyDirect uses wc_AesSetKey internally. **Warning:** In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB API's directly whenever possible.
int	**wc_AesGcmSetKey**(Aes * aes, const byte * key, word32 len)This function is used to set the key for AES GCM (Galois/Counter Mode). It initializes an AES object with the given key. It is only enabled if the configure option HAVE_AESGCM is enabled at compile time.

	Name
int	**wc_AesSetTagLen**(Aes * aes, word32 tagLen)This function associates one tag length with the key held by an AES object, as SP 800_38D section 5.2.1.2 and SP 800_38C section 5.3 require. Once set, AES_GCM and AES_CCM calls with that key accept only that tag length. wc_AesGcmSetKey() and wc_AesCcmSetKey() clear the association. The mode still applies its own list of allowed tag lengths on top of this.
int	**wc_AesGcmEncrypt**(Aes * aes, byte * out, const byte * in, word32 sz, const byte * iv, word32 ivSz, byte * authTag, word32 authTagSz, const byte * authIn, word32 authInSz)This function encrypts the input message, held in the buffer in, and stores the resulting cipher text in the output buffer out. It requires a new iv (initialization vector) for each call to encrypt. It also encodes the input authentication vector, authIn, into the authentication tag, authTag.
int	**wc_AesGcmDecrypt**(Aes * aes, byte * out, const byte * in, word32 sz, const byte * iv, word32 ivSz, const byte * authTag, word32 authTagSz, const byte * authIn, word32 authInSz)This function decrypts the input cipher text, held in the buffer in, and stores the resulting message text in the output buffer out. It also checks the input authentication vector, authIn, against the supplied authentication tag, authTag. If a nonzero error code is returned, the output data is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.
int	**wc_GmacSetKey**(Gmac * gmac, const byte * key, word32 len)This function initializes and sets the key for a GMAC object to be used for Galois Message Authentication.
int	**wc_GmacUpdate**(Gmac * gmac, const byte * iv, word32 ivSz, const byte * authIn, word32 authInSz, byte * authTag, word32 authTagSz)This function generates the Gmac hash of the authIn input and stores the result in the authTag buffer. After running wc_GmacUpdate, one should compare the generated authTag to a known authentication tag to verify the authenticity of a message.
int	**wc_AesCcmSetKey**(Aes * aes, const byte * key, word32 keySz)This function sets the key for an AES object using CCM (Counter with CBC_MAC). It takes a pointer to an AES structure and initializes it with supplied key.

	Name
int	**wc_AesCcmEncrypt**(Aes * aes, byte * out, const byte * in, word32 inSz, const byte * nonce, word32 nonceSz, byte * authTag, word32 authTagSz, const byte * authIn, word32 authInSz)This function encrypts the input message, in, into the output buffer, out, using CCM (Counter with CBC_MAC). It subsequently calculates and stores the authorization tag, authTag, from the authIn input.
int	**wc_AesCcmDecrypt**(Aes * aes, byte * out, const byte * in, word32 inSz, const byte * nonce, word32 nonceSz, const byte * authTag, word32 authTagSz, const byte * authIn, word32 authInSz)This function decrypts the input cipher text, in, into the output buffer, out, using CCM (Counter with CBC_MAC). It subsequently calculates the authorization tag, authTag, from the authIn input. If a nonzero error code is returned, the output data is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.
int	**wc_AesXtsInit**(XtsAes * aes, void * heap, int devId)This is to initialize an AES-XTS context. It is up to user to call wc_AesXtsFree on aes key when done.
int	**wc_AesXtsSetKeyNoInit**(XtsAes * aes, const byte * key, word32 len, int dir)This is to help with setting keys to correct encrypt or decrypt type, after first calling wc_AesXtsInit(). It is up to user to call wc_AesXtsFree on aes key when done.
int	**wc_AesXtsSetKey**(XtsAes * aes, const byte * key, word32 len, int dir, void * heap, int devId)This is to help with setting keys to correct encrypt or decrypt type. It is up to user to call wc_AesXtsFree on aes key when done.
int	**wc_AesXtsEncryptSector**(XtsAes * aes, byte * out, const byte * in, word32 sz, word64 sector)Same process as wc_AesXtsEncrypt but uses a word64 type as the tweak value instead of a byte array. This just converts the word64 to a byte array and calls wc_AesXtsEncrypt.
int	**wc_AesXtsDecryptSector**(XtsAes * aes, byte * out, const byte * in, word32 sz, word64 sector)Same process as wc_AesXtsDecrypt but uses a word64 type as the tweak value instead of a byte array. This just converts the word64 to a byte array.

	Name
int	**wc_AesXtsEncrypt**(XtsAes * aes, byte * out, const byte * in, word32 sz, const byte * i, word32 iSz)AES with XTS mode. (XTS) XEX encryption with Tweak and cipher text Stealing.
int	**wc_AesXtsDecrypt**(XtsAes * aes, byte * out, const byte * in, word32 sz, const byte * i, word32 iSz)Same process as encryption but Aes key is AES_DECRYPTION type.
int	**wc_AesXtsFree**(XtsAes * aes)This is to free up any resources used by the XtsAes structure.
int	**wc_AesInit**(Aes * aes, void * heap, int devId)Initialize Aes structure. Sets heap hint to be used and ID for use with async hardware. It is up to the user to call wc_AesFree on the Aes structure when done.
void	**wc_AesFree**(Aes * aes)free resources associated with the Aes structure when applicable. Internally may sometimes be a no_op but still recommended to call in all cases as a general best_practice (IE if application code is ported for use on new environments where the call is applicable).
int	**wc_AesCfbEncrypt**(Aes * aes, byte * out, const byte * in, word32 sz)AES with CFB mode.
int	**wc_AesCfbDecrypt**(Aes * aes, byte * out, const byte * in, word32 sz)AES with CFB mode.
int	**wc_AesSivEncrypt**(const byte * key, word32 keySz, const byte * assoc, word32 assocSz, const byte * nonce, word32 nonceSz, const byte * in, word32 inSz, byte * siv, byte * out)This function performs SIV (synthetic initialization vector) encryption as described in RFC 5297.
int	**wc_AesSivDecrypt**(const byte * key, word32 keySz, const byte * assoc, word32 assocSz, const byte * nonce, word32 nonceSz, const byte * in, word32 inSz, byte * siv, byte * out)This function performs SIV (synthetic initialization vector) decryption as described in RFC 5297. If a nonzero error code is returned, the output data is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.

	Name
WOLFSSL_API int	**wc_AesEaxEncryptAuth**(const byte * key, word32 keySz, byte * out, const byte * in, word32 inSz, const byte * nonce, word32 nonceSz, byte * authTag, word32 authTagSz, const byte * authIn, word32 authInSz)This function performs AES EAX encryption and authentication as described in “EAX: A Conventional Authenticated_Encryption Mode” (https://eprint.iacr.org/2003/069). It is a “one-shot” API that performs all encryption and authentication operations in one function call.
WOLFSSL_API int	**wc_AesEaxDecryptAuth**(const byte * key, word32 keySz, byte * out, const byte * in, word32 inSz, const byte * nonce, word32 nonceSz, const byte * authTag, word32 authTagSz, const byte * authIn, word32 authInSz)This function performs AES EAX decryption and authentication as described in “EAX: A Conventional Authenticated_Encryption Mode” (https://eprint.iacr.org/2003/069). It is a “one-shot” API that performs all decryption and authentication operations in one function call. If a nonzero error code is returned, the output data is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.
WOLFSSL_API int	**wc_AesEaxInit**(AesEax * eax, const byte * key, word32 keySz, const byte * nonce, word32 nonceSz, const byte * authIn, word32 authInSz)This function initializes an AesEax object for use in authenticated encryption or decryption. This function must be called on an AesEax object before using it with any of the AES EAX incremental API functions. It does not need to be called if using the one_shot EAX API functions. All AesEax instances initialized with this function need to be freed with a call to wc_AesEaxFree() when done using the instance.
WOLFSSL_API int	**wc_AesEaxEncryptUpdate**(AesEax * eax, byte * out, const byte * in, word32 inSz, const byte * authIn, word32 authInSz)This function uses AES EAX to encrypt input data, and optionally, add more input data to the authentication stream. eax must have been previously initialized with a call to wc_AesEaxInit.
WOLFSSL_API int	**wc_AesEaxDecryptUpdate**(AesEax * eax, byte * out, const byte * in, word32 inSz, const byte * authIn, word32 authInSz)This function uses AES EAX to decrypt input data, and optionally, add more input data to the authentication stream. eax must have been previously initialized with a call to wc_AesEaxInit.

	Name
WOLFSSL_API int	**wc_AesEaxAuthDataUpdate**(AesEax * eax, const byte * authIn, word32 authInSz)This function adds input data to the authentication stream. eax must have been previously initialized with a call to wc_AesEaxInit.
WOLFSSL_API int	**wc_AesEaxEncryptFinal**(AesEax * eax, byte * authTag, word32 authTagSz)This function finalizes the encrypt AEAD operation, producing an auth tag over the current authentication stream. eax must have been previously initialized with a call to wc_AesEaxInit. When done using the AesEax context structure, make sure to free it using wc_AesEaxFree.
WOLFSSL_API int	**wc_AesEaxDecryptFinal**(AesEax * eax, const byte * authIn, word32 authInSz)This function finalizes the decrypt AEAD operation, finalizing the auth tag computation and checking it for validity against the user supplied tag. eax must have been previously initialized with a call to wc_AesEaxInit. When done using the AesEax context structure, make sure to free it using wc_AesEaxFree.
WOLFSSL_API int	**wc_AesEaxFree**(AesEax * eax)This frees up any resources, specifically keys, used by the Aes instance inside the AesEax wrapper struct. It should be called on the AesEax struct after it has been initialized with wc_AesEaxInit, and all desired EAX operations are complete.
int	**wc_AesCtsEncrypt**(const byte * key, word32 keySz, byte * out, const byte * in, word32 inSz, const byte * iv)This function performs AES encryption using Ciphertext Stealing (CTS) mode. It is a one-shot API that handles all operations in a single call.
int	**wc_AesCtsDecrypt**(const byte * key, word32 keySz, byte * out, const byte * in, word32 inSz, const byte * iv)This function performs AES decryption using Ciphertext Stealing (CTS) mode. It is a one-shot API that handles all operations in a single call.
int	**wc_AesCtsEncryptUpdate**(Aes * aes, byte * out, word32 * outSz, const byte * in, word32 inSz)This function performs an update step of the AES CTS encryption. It processes a chunk of plaintext and stores intermediate data.
int	**wc_AesCtsEncryptFinal**(Aes * aes, byte * out, word32 * outSz)This function finalizes the AES CTS encryption operation. It processes any remaining plaintext and completes the encryption.

	Name
int	**wc_AesCtsDecryptUpdate**(Aes * aes, byte * out, word32 * outSz, const byte * in, word32 inSz)This function performs an update step of the AES CTS decryption. It processes a chunk of ciphertext and stores intermediate data.
int	**wc_AesCtsDecryptFinal**(Aes * aes, byte * out, word32 * outSz)This function finalizes the AES CTS decryption operation. It processes any remaining ciphertext and completes the decryption.
int	**wc_AesCfb1Encrypt**(Aes * aes, byte * out, const byte * in, word32 sz)This function encrypts data using AES CFB_1 mode (1_bit feedback). It processes data one bit at a time, making it suitable for bit-oriented applications.
int	**wc_AesCfb8Encrypt**(Aes * aes, byte * out, const byte * in, word32 sz)This function encrypts data using AES CFB_8 mode (8_bit feedback). It processes data one byte at a time, making it suitable for byte-oriented stream encryption.
int	**wc_AesCfb1Decrypt**(Aes * aes, byte * out, const byte * in, word32 sz)This function decrypts data using AES CFB_1 mode (1_bit feedback). It processes data one bit at a time, making it suitable for bit-oriented applications.
int	**wc_AesCfb8Decrypt**(Aes * aes, byte * out, const byte * in, word32 sz)This function decrypts data using AES CFB_8 mode (8_bit feedback). It processes data one byte at a time, making it suitable for byte-oriented stream decryption.
int	**wc_AesOfbEncrypt**(Aes * aes, byte * out, const byte * in, word32 sz)This function encrypts data using AES OFB mode (Output Feedback). OFB mode turns a block cipher into a stream cipher by encrypting the IV and XORing with plaintext.
int	**wc_AesOfbDecrypt**(Aes * aes, byte * out, const byte * in, word32 sz)This function decrypts data using AES OFB mode (Output Feedback). In OFB mode, encryption and decryption are the same operation.
int	**wc_AesEcbEncrypt**(Aes * aes, byte * out, const byte * in, word32 sz)This function encrypts data using AES ECB mode (Electronic Codebook). Warning: ECB mode is not recommended for most use cases as it does not provide semantic security. Each block is encrypted independently.

	Name
int	**wc_AesEcbDecrypt**(Aes * aes, byte * out, const byte * in, word32 sz)This function decrypts data using AES ECB mode (Electronic Codebook). Warning: ECB mode is not recommended for most use cases as it does not provide semantic security. Each block is decrypted independently.
int	**wc_AesCtrSetKey**(Aes * aes, const byte * key, word32 len, const byte * iv, int dir)This function sets the key and IV for AES CTR mode. It initializes the AES structure for counter mode encryption or decryption.
int	**wc_AesGcmSetKey_ex**(Aes * aes, const byte * key, word32 len, word32 kup)This function sets the key for AES GCM with an extended key update parameter. It allows for key updates in certain hardware implementations.
int	**wc_AesGcmInit**(Aes * aes, const byte * key, word32 len, const byte * iv, word32 ivSz)This function initializes an AES GCM cipher with key and IV. It can be called with NULL key to only set the IV, or with NULL IV to only set the key.
int	**wc_AesGcmEncryptInit**(Aes * aes, const byte * key, word32 len, const byte * iv, word32 ivSz)This function initializes an AES GCM cipher for encryption. It is a convenience wrapper around wc_AesGcmInit for encryption operations.
int	**wc_AesGcmEncryptInit_ex**(Aes * aes, const byte * key, word32 len, byte * ivOut, word32 ivOutSz)This function initializes an AES GCM cipher for encryption and outputs the IV. This is useful when part of the IV is generated internally. Must call wc_AesGcmSetIV() before this function to set the fixed part of the IV.
int	**wc_AesGcmEncryptUpdate**(Aes * aes, byte * out, const byte * in, word32 sz, const byte * authIn, word32 authInSz)This function performs an update step of AES GCM encryption. It processes plaintext and/or additional authentication data (AAD) in a streaming fashion.
int	**wc_AesGcmEncryptFinal**(Aes * aes, byte * authTag, word32 authTagSz)This function finalizes AES GCM encryption and generates the authentication tag. This must be called after all data has been processed with wc_AesGcmEncryptUpdate.

	Name
int	**wc_AesGcmDecryptInit**(Aes * aes, const byte * key, word32 len, const byte * iv, word32 ivSz)This function initializes an AES GCM cipher for decryption. It is a convenience wrapper around wc_AesGcmInit for decryption operations.
int	**wc_AesGcmDecryptUpdate**(Aes * aes, byte * out, const byte * in, word32 sz, const byte * authIn, word32 authInSz)This function performs an update step of AES GCM decryption. It processes ciphertext and/or additional authentication data (AAD) in a streaming fashion.
int	**wc_AesGcmDecryptFinal**(Aes * aes, const byte * authTag, word32 authTagSz)This function finalizes AES GCM decryption and verifies the authentication tag. This must be called after all data has been processed with wc_AesGcmDecryptUpdate.
int	**wc_AesGcmSetExtIV**(Aes * aes, const byte * iv, word32 ivSz)This function sets an external IV for AES GCM. This allows using an IV that was generated externally or received from another source.
int	**wc_AesGcmSetIV**(Aes * aes, word32 ivSz, const byte * ivFixed, word32 ivFixedSz, WC_RNG * rng)This function sets the IV for AES GCM with optional random generation. It can generate part of the IV using an RNG, which is useful for ensuring IV uniqueness.
int	**wc_AesGcmEncrypt_ex**(Aes * aes, byte * out, const byte * in, word32 sz, byte * ivOut, word32 ivOutSz, byte * authTag, word32 authTagSz, const byte * authIn, word32 authInSz)This function performs AES GCM encryption with extended parameters, including IV output. This is a one-shot encryption function that outputs the generated IV.
int	**wc_Gmac**(const byte * key, word32 keySz, byte * iv, word32 ivSz, const byte * authIn, word32 authInSz, byte * authTag, word32 authTagSz, WC_RNG * rng)This function performs GMAC (Galois Message Authentication Code) generation. GMAC is essentially AES-GCM with no plaintext, used for authentication only.
int	**wc_GmacVerify**(const byte * key, word32 keySz, const byte * iv, word32 ivSz, const byte * authIn, word32 authInSz, const byte * authTag, word32 authTagSz)This function verifies a GMAC (Galois Message Authentication Code). It computes the GMAC and compares it with the provided tag.

	Name
int	**wc_AesCcmSetNonce**(Aes * aes, const byte * nonce, word32 nonceSz)This function sets the nonce for AES CCM mode. The nonce must be set before encryption or decryption operations.
int	**wc_AesCcmEncrypt_ex**(Aes * aes, byte * out, const byte * in, word32 sz, byte * ivOut, word32 ivOutSz, byte * authTag, word32 authTagSz, const byte * authIn, word32 authInSz)This function performs AES CCM encryption with extended parameters, including nonce output. This is useful when part of the nonce is generated internally.
int	**wc_AesKeyWrap**(const byte * key, word32 keySz, const byte * in, word32 inSz, byte * out, word32 outSz, const byte * iv)This function wraps a key using AES Key Wrap algorithm (RFC 3394). This is commonly used to securely transport cryptographic keys.
int	**wc_AesKeyWrap_ex**(Aes * aes, const byte * in, word32 inSz, byte * out, word32 outSz, const byte * iv)This function wraps a key using AES Key Wrap algorithm with an initialized AES structure. This allows reusing the same AES structure for multiple wrap operations.
int	**wc_AesKeyUnWrap**(const byte * key, word32 keySz, const byte * in, word32 inSz, byte * out, word32 outSz, const byte * iv)This function unwraps a key using AES Key Unwrap algorithm (RFC 3394). This is used to securely receive cryptographic keys that were wrapped.
int	**wc_AesKeyUnWrap_ex**(Aes * aes, const byte * in, word32 inSz, byte * out, word32 outSz, const byte * iv)This function unwraps a key using AES Key Unwrap algorithm with an initialized AES structure. This allows reusing the same AES structure for multiple unwrap operations.
int	**wc_AesXtsEncryptConsecutiveSectors**(XtsAes * aes, byte * out, const byte * in, word32 sz, word64 sector, word32 sectorSz)This function encrypts multiple consecutive sectors using AES XTS mode. It processes multiple sectors in sequence, automatically incrementing the sector number for each sector.
int	**wc_AesXtsDecryptConsecutiveSectors**(XtsAes * aes, byte * out, const byte * in, word32 sz, word64 sector, word32 sectorSz)This function decrypts multiple consecutive sectors using AES XTS mode. It processes multiple sectors in sequence, automatically incrementing the sector number for each sector.

	Name
int	**wc_AesXtsEncryptInit**(XtsAes * aes, const byte * i, word32 iSz, struct XtsAesStreamData * stream)This function initializes streaming AES XTS encryption. It sets up the context for processing data in multiple update calls.
int	**wc_AesXtsDecryptInit**(XtsAes * aes, const byte * i, word32 iSz, struct XtsAesStreamData * stream)This function initializes streaming AES XTS decryption. It sets up the context for processing data in multiple update calls.
int	**wc_AesXtsEncryptUpdate**(XtsAes * aes, byte * out, const byte * in, word32 sz, struct XtsAesStreamData * stream)This function performs an update step of streaming AES XTS encryption. It processes a chunk of data and can be called multiple times.
int	**wc_AesXtsDecryptUpdate**(XtsAes * aes, byte * out, const byte * in, word32 sz, struct XtsAesStreamData * stream)This function performs an update step of streaming AES XTS decryption. It processes a chunk of data and can be called multiple times.
int	**wc_AesXtsEncryptFinal**(XtsAes * aes, byte * out, const byte * in, word32 sz, struct XtsAesStreamData * stream)This function finalizes streaming AES XTS encryption. It processes any remaining data and completes the encryption operation.
int	**wc_AesXtsDecryptFinal**(XtsAes * aes, byte * out, const byte * in, word32 sz, struct XtsAesStreamData * stream)This function finalizes streaming AES XTS decryption. It processes any remaining data and completes the decryption operation.
int	**wc_AesGetKeySize**(Aes * aes, word32 * keySize)This function retrieves the key size from an initialized AES structure. It returns the size of the key currently set in the AES object.
int	**wc_AesInit_Id**(Aes * aes, unsigned char * id, int len, void * heap, int devId)This function initializes an AES structure with an ID. This is useful for tracking or identifying specific AES instances in applications that manage multiple AES contexts.
int	**wc_AesInit_Label**(Aes * aes, const char * label, void * heap, int devId)This function initializes an AES structure with a label string. This is useful for tracking or identifying specific AES instances with human-readable names.

	Name
Aes *	**wc_AesNew**(void * heap, int devId, int * result_code)This function allocates and initializes a new AES structure. It returns a pointer to the allocated structure, which must be freed with wc_AesDelete when no longer needed. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.
int	**wc_AesDelete**(Aes * aes, Aes ** aes_p)This function frees an AES structure that was allocated with wc_AesNew. It also sets the pointer to NULL to prevent use-after-free. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.
int	**wc_AesSivEncrypt_ex**(const byte * key, word32 keySz, const AesSivAssoc * assoc, word32 numAssoc, const byte * nonce, word32 nonceSz, const byte * in, word32 inSz, byte * siv, byte * out)This function performs AES_SIV (Synthetic IV) encryption with extended parameters. AES-SIV provides nonce-misuse resistance and deterministic authenticated encryption.
int	**wc_AesSivDecrypt_ex**(const byte * key, word32 keySz, const AesSivAssoc * assoc, word32 numAssoc, const byte * nonce, word32 nonceSz, const byte * in, word32 inSz, byte * siv, byte * out)This function performs AES_SIV (Synthetic IV) decryption with extended parameters. It verifies the SIV and decrypts the ciphertext.
int	**wc_AesCbcDecryptWithKey**(byte * out, const byte * in, word32 inSz, const byte * key, word32 keySz, const byte * iv)Decrypts a cipher from the input buffer in, and places the resulting plain text in the output buffer out using cipher block chaining with AES. This function does not require an AES structure to be initialized. Instead, it takes in a key and an iv (initialization vector) and uses these to initialize an AES object and then decrypt the cipher text.

	Name
int	**wc_AesCbcEncryptWithKey**(byte * out, const byte * in, word32 inSz, const byte * key, word32 keySz, const byte * iv)This function encrypts a plaintext message and stores the result in the output buffer. It uses AES encryption with cipher block chaining (CBC) mode. This function does not require an AES structure to be initialized. Instead, it takes in a key and an iv and uses these to encrypt the message.

B.12.2 Functions Documentation

```
int wc_AesSetKey(
    Aes * aes,
    const byte * key,
    word32 len,
    const byte * iv,
    int dir
)
```

This function initializes an AES structure by setting the key and then setting the initialization vector.

Parameters:

- **aes** pointer to the AES structure to modify
- **key** 16, 24, or 32 byte secret key for encryption and decryption
- **len** length of the key passed in
- **iv** pointer to the initialization vector used to initialize the key
- **dir** Cipher direction. Set AES_ENCRYPTION to encrypt, or AES_DECRYPTION to decrypt. Direction for some modes (CFB and CTR) is always AES_ENCRYPTION.

See:

- wc_AesSetKeyDirect
- wc_AesSetIV

Return:

- 0 On successfully setting key and initialization vector.
- BAD_FUNC_ARG Returned if key length is invalid.

Example

```
Aes enc;
int ret = 0;
byte key[] = { some 16, 24 or 32 byte key };
byte iv[]  = { some 16 byte iv };
if (ret = wc_AesInit(&enc, HEAP_HINT, INVALID_DEVID) != 0) {
    // failed to initialize aes key
}
```

```
if (ret = wc_AesSetKey(&enc, key, AES_BLOCK_SIZE, iv,
AES_ENCRYPTION) != 0) {
// failed to set aes key
}
```

```
int wc_AesSetIV(
    Aes * aes,
    const byte * iv
)
```

This function sets the initialization vector for a particular AES object. The AES object should be initialized before calling this function.

Parameters:

- **aes** pointer to the AES structure on which to set the initialization vector
- **iv** initialization vector used to initialize the AES structure. If the value is NULL, the default action initializes the iv to 0.

See:

- wc_AesSetKeyDirect
- wc_AesSetKey

Return:

- 0 On successfully setting initialization vector.
- BAD_FUNC_ARG Returned if AES pointer is NULL.

Example

```
Aes enc;
// set enc key
byte iv[]  = { some 16 byte iv };
if (ret = wc_AesSetIV(&enc, iv) != 0) {
// failed to set aes iv
}
```

```
int wc_AesCbcEncrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

Encrypts a plaintext message from the input buffer in, and places the resulting cipher text in the output buffer out using cipher block chaining with AES. This function requires that the AES object has been initialized by calling AesSetKey before a message is able to be encrypted. This function assumes that

the input message is AES block length aligned, and expects the input length to be a multiple of the block length, which will optionally be checked and enforced if WOLFSSL_AES_CBC_LENGTH_CHECKS is defined in the build configuration. In order to assure block-multiple input, PKCS#7 style padding should be added beforehand. This differs from the OpenSSL AES-CBC methods which add the padding for you. To make the wolfSSL and corresponding OpenSSL functions interoperate, one should specify the -nopad option in the OpenSSL command line function so that it behaves like the wolfSSL AesCbcEncrypt method and does not add extra padding during encryption.

Parameters:

- **aes** pointer to the AES object used to encrypt data
- **out** pointer to the output buffer in which to store the ciphertext of the encrypted message
- **in** pointer to the input buffer containing message to be encrypted
- **sz** size of input message

See:

- wc_AesInit
- wc_AesSetKey
- wc_AesSetIV
- wc_AesCbcDecrypt

Return:

- 0 On successfully encrypting message.
- BAD_ALIGN_E: may be returned on block align error
- BAD_LENGTH_E will be returned if the input length isn't a multiple of the AES block length, when the library is built with WOLFSSL_AES_CBC_LENGTH_CHECKS.

Example

```
Aes enc;
int ret = 0;
// initialize enc with wc_AesInit and wc_AesSetKey, using direction
// AES_ENCRYPTION
byte msg[AES_BLOCK_SIZE * n]; // multiple of 16 bytes
// fill msg with data
byte cipher[AES_BLOCK_SIZE * n]; // Some multiple of 16 bytes
if ((ret = wc_AesCbcEncrypt(&enc, cipher, message, sizeof(msg))) != 0 ) {
// block align error
}
```

```
int wc_AesCbcDecrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

Decrypts a cipher from the input buffer in, and places the resulting plain text in the output buffer out using cipher block chaining with AES. This function requires that the AES structure has been initialized by calling AesSetKey before a message is able to be decrypted. This function assumes that the original message was AES block length aligned, and expects the input length to be a multiple of the block length, which will optionally be checked and enforced if WOLFSSL_AES_CBC_LENGTH_CHECKS is defined in the build configuration. This differs from the OpenSSL AES-CBC methods, which add PKCS#7 padding automatically, and so do not require block-multiple input. To make the wolfSSL function and equivalent OpenSSL functions interoperate, one should specify the -nopad option in the OpenSSL command line function so that it behaves like the wolfSSL AesCbcEncrypt method and does not create errors during decryption.

Parameters:

- **aes** pointer to the AES object used to decrypt data.
- **out** pointer to the output buffer in which to store the plain text of the decrypted message. size must be a multiple of AES_BLOCK_LENGTH, padded if necessary
- **in** pointer to the input buffer containing cipher text to be decrypted. size must be a multiple of AES_BLOCK_LENGTH, padded if necessary
- **sz** size of input message.

See:

- wc_AesInit
- wc_AesSetKey
- wc_AesCbcEncrypt

Return:

- 0 On successfully decrypting message.
- BAD_ALIGN_E may be returned on block align error.
- BAD_LENGTH_E will be returned if the input length isn't a multiple of the AES block length, when the library is built with WOLFSSL_AES_CBC_LENGTH_CHECKS.

Example

```
Aes dec;
int ret = 0;
// initialize dec with wc_AesInit and wc_AesSetKey, using direction
// AES_DECRYPTION
byte cipher[AES_BLOCK_SIZE * n]; // some multiple of 16 bytes
// fill cipher with cipher text
byte plain [AES_BLOCK_SIZE * n];
if ((ret = wc_AesCbcDecrypt(&dec, plain, cipher, sizeof(cipher))) != 0 ) {
// block align error
}
```

```
int wc_AesCtrEncrypt(
    Aes * aes,
    byte * out,
    const byte * in,
```

```
    word32 sz
)
```

Encrypts/Decrypts a message from the input buffer in, and places the resulting cipher text in the output buffer out using CTR mode with AES. This function is only enabled if WOLFSSL_AES_COUNTER is enabled at compile time. The AES structure should be initialized through AesSetKey before calling this function. Note that this function is used for both decryption and encryption. *NOTE:* Regarding using same API for encryption and decryption. User should differentiate between Aes structures for encrypt/decrypt.

Parameters:

- **aes** pointer to the AES object used to decrypt data
- **out** pointer to the output buffer in which to store the cipher text of the encrypted message size must be a multiple of AES_BLOCK_LENGTH, padded if necessary
- **in** pointer to the input buffer containing plain text to be encrypted size must be a multiple of AES_BLOCK_LENGTH, padded if necessary
- **sz** size of the input plain text

See: wc_AesSetKey

Return: int integer values corresponding to wolfSSL error or success status

Example

```
Aes enc;
Aes dec;
// initialize enc and dec with wc_AesInit and wc_AesSetKeyDirect, using
// direction AES_ENCRYPTION since the underlying API only calls Encrypt
// and by default calling encrypt on a cipher results in a decryption of
// the cipher

byte msg[AES_BLOCK_SIZE * n]; //n being a positive integer making msg
some multiple of 16 bytes
// fill plain with message text
byte cipher[AES_BLOCK_SIZE * n];
byte decrypted[AES_BLOCK_SIZE * n];
wc_AesCtrEncrypt(&enc, cipher, msg, sizeof(msg)); // encrypt plain
wc_AesCtrEncrypt(&dec, decrypted, cipher, sizeof(cipher));
// decrypt cipher text
```

```
int wc_AesEncryptDirect(
    Aes * aes,
    byte * out,
    const byte * in
)
```

This function is a one-block encrypt of the input block, in, into the output block, out. It uses the key of the provided AES structure, which should be initialized with wc_AesSetKey before calling this function. wc_AesSetKey should have been called with the iv set to NULL. This is only enabled if the configure option WOLFSSL_AES_DIRECT is enabled. **Warning:** In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB API's directly whenever possible. When

WC_AES_DIRECT_CRYPTO_CB is defined (set automatically for WOLF_CRYPTO_CB builds that use the generic AES Direct code, not a port's own version), if the AES object has a crypto callback device (devId), that device is asked first to encrypt the block as a one-block ECB request. The software key is used only if the device declines, and a device error is returned as-is. A device that owns the key must handle the request or return an error; it must not return CRYPTOCB_UNAVAILABLE. With WOLF_CRYPTO_CB_FIND, the device find callback picks the device, so a device can be asked even when the AES object has no devId.

Parameters:

- **aes** pointer to the AES object used to encrypt data
- **out** pointer to the output buffer in which to store the cipher text of the encrypted message
- **in** pointer to the input buffer containing plain text to be encrypted

See:

- wc_AesDecryptDirect
- wc_AesSetKeyDirect

Return: int integer values corresponding to wolfSSL error or success status

Example

```
Aes enc;
// initialize enc with wc_AesInit and wc_AesSetKey, using direction
// AES_ENCRYPTION
byte msg [AES_BLOCK_SIZE]; // 16 bytes
// initialize msg with plain text to encrypt
byte cipher[AES_BLOCK_SIZE];
wc_AesEncryptDirect(&enc, cipher, msg);
```

```
int wc_AesDecryptDirect(
    Aes * aes,
    byte * out,
    const byte * in
)
```

This function is a one-block decrypt of the input block, in, into the output block, out. It uses the key of the provided AES structure, which should be initialized with wc_AesSetKey before calling this function. wc_AesSetKey should have been called with the iv set to NULL. This is only enabled if the configure option WOLFSSL_AES_DIRECT is enabled. **Warning:** In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB API's directly whenever possible. When WC_AES_DIRECT_CRYPTO_CB is defined (set automatically for WOLF_CRYPTO_CB builds that use the generic AES Direct code, not a port's own version), if the AES object has a crypto callback device (devId), that device is asked first to decrypt the block as a one-block ECB request. The software key is used only if the device declines, and a device error is returned as-is. A device that owns the key must handle the request or return an error; it must not return CRYPTOCB_UNAVAILABLE. With WOLF_CRYPTO_CB_FIND, the device find callback picks the device, so a device can be asked even when the AES object has no devId.

Parameters:

- **aes** pointer to the AES object used to encrypt data
- **out** pointer to the output buffer in which to store the plain text of the decrypted cipher text
- **in** pointer to the input buffer containing cipher text to be decrypted

See:

- wc_AesEncryptDirect
- wc_AesSetKeyDirect

Return: int integer values corresponding to wolfSSL error or success status

Example

```
Aes dec;
// initialize enc with wc_AesInit and wc_AesSetKey, using direction
// AES_DECRYPTION
byte cipher [AES_BLOCK_SIZE]; // 16 bytes
// initialize cipher with cipher text to decrypt
byte msg[AES_BLOCK_SIZE];
wc_AesDecryptDirect(&dec, msg, cipher);
```

```
int wc_AesSetKeyDirect(
    Aes * aes,
    const byte * key,
    word32 len,
    const byte * iv,
    int dir
)
```

This function is used to set the AES keys for CTR mode with AES. It initializes an AES object with the given key, iv (initialization vector), and encryption dir (direction). It is only enabled if the configure option WOLFSSL_AES_DIRECT is enabled. Currently wc_AesSetKeyDirect uses wc_AesSetKey internally. **Warning:** In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB API's directly whenever possible.

Parameters:

- **aes** pointer to the AES object used to encrypt data
- **key** 16, 24, or 32 byte secret key for encryption and decryption
- **len** length of the key passed in
- **iv** initialization vector used to initialize the key
- **dir** Cipher direction. Set AES_ENCRYPTION to encrypt, or AES_DECRYPTION to decrypt. (See enum in wolfssl/wolfcrypt/aes.h) (NOTE: If using wc_AesSetKeyDirect with Aes Counter mode (Stream cipher) only use AES_ENCRYPTION for both encrypting and decrypting)

See:

- wc_AesEncryptDirect
- wc_AesDecryptDirect

- wc_AesSetKey

Return:

- 0 On successfully setting the key.
- BAD_FUNC_ARG Returned if the given key is an invalid length.

Example

```
Aes enc;
int ret = 0;
byte key[] = { some 16, 24, or 32 byte key };
byte iv[]  = { some 16 byte iv };

if (ret = wc_AesInit(&enc, HEAP_HINT, INVALID_DEVID) != 0) {
    // failed to initialize aes key
}
if (ret = wc_AesSetKeyDirect(&enc, key, sizeof(key), iv,
AES_ENCRYPTION) != 0) {
// failed to set aes key
}
```

```
int wc_AesGcmSetKey(
    Aes * aes,
    const byte * key,
    word32 len
)
```

This function is used to set the key for AES GCM (Galois/Counter Mode). It initializes an AES object with the given key. It is only enabled if the configure option HAVE_AESGCM is enabled at compile time.

Parameters:

- **aes** pointer to the AES object used to encrypt data
- **key** 16, 24, or 32 byte secret key for encryption and decryption
- **len** length of the key passed in

See:

- wc_AesGcmEncrypt
- wc_AesGcmDecrypt

Return:

- 0 On successfully setting the key.
- BAD_FUNC_ARG Returned if the given key is an invalid length.

Example

```
Aes enc;
int ret = 0;
byte key[] = { some 16, 24,32 byte key };
if (ret = wc_AesInit(&enc, HEAP_HINT, INVALID_DEVID) != 0) {
    // failed to initialize aes key
}
if (ret = wc_AesGcmSetKey(&enc, key, sizeof(key)) != 0) {
// failed to set aes key
}
```

```
int wc_AesSetTagLen(
    Aes * aes,
    word32 tagLen
)
```

This function associates one tag length with the key held by an AES object, as SP 800-38D section 5.2.1.2 and SP 800-38C section 5.3 require. Once set, AES-GCM and AES-CCM calls with that key accept only that tag length. wc_AesGcmSetKey() and wc_AesCcmSetKey() clear the association. The mode still applies its own list of allowed tag lengths on top of this.

Parameters:

- **aes** pointer to the AES object holding the key
- **tagLen** tag length to associate with the key

See:

- wc_AesGcmSetKey
- wc_AesCcmSetKey
- wc_CmacSetTagLen

Return:

- 0 On success.
- BAD_FUNC_ARG Returned if aes is NULL, the length is zero or larger than the AES block size, or a different length is already associated.

WOLFSSL_NO_AES_TAG_ASSOCIATION builds all of this out, which departs from the requirement above, so a FIPS v7 or later build refuses it.

The first use of a key writes the length into the AES object, so a keyed object shared read only between threads for one-shot AES-GCM or AES-CCM needs its length set here before it is handed out.

The first use of a key fixes its tag length even without this call. Once a length is associated this call can repeat it but not change or clear it; set the key again through one of those two to start over. wc_AesSetKey() clears it as well on builds that use the software key schedule, but a backend with its own key setter does not. Hardware offload ports do not apply this association, and such a build would be validated as a hybrid module.

A crypto callback is checked here before the work is handed over, because the key it forwards is the one this object holds. A callback that builds a fresh context per call cannot see the association, so this is the only place it can be caught. A device key installed by id or label is the exception, it is never seen here, so its first tag length stays associated until the object is freed.

An OpenSSL-compat EVP context re-initialized without a new key keeps the association, so a later verify asking for a different tag length fails. That is the one key one tag length rule, not a caller error. EVP encrypt always tags at the full length and shortens afterwards, so it never reaches this check.

Example

```
Aes enc;
byte key[] = { some 16, 24, 32 byte key };
if (wc_AesGcmSetKey(&enc, key, sizeof(key)) != 0) {
    // failed to set aes key
}
if (wc_AesSetTagLen(&enc, 16) != 0) {
    // failed to associate the tag length
}
```

```
int wc_AesGcmEncrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    const byte * iv,
    word32 ivSz,
    byte * authTag,
    word32 authTagSz,
    const byte * authIn,
    word32 authInSz
)
```

This function encrypts the input message, held in the buffer in, and stores the resulting cipher text in the output buffer out. It requires a new iv (initialization vector) for each call to encrypt. It also encodes the input authentication vector, authIn, into the authentication tag, authTag.

Parameters:

- **aes** - pointer to the AES object used to encrypt data
- **out** pointer to the output buffer in which to store the cipher text size must match in's size (sz)
- **in** pointer to the input buffer holding the message to encrypt size must be a multiple of AES_BLOCK_LENGTH, padded if necessary
- **sz** length of the input message to encrypt
- **iv** pointer to the buffer containing the initialization vector
- **ivSz** length of the initialization vector
- **authTag** pointer to the buffer in which to store the authentication tag
- **authTagSz** length of the desired authentication tag
- **authIn** pointer to the buffer containing the input authentication vector
- **authInSz** length of the input authentication vector

See:

- wc_AesGcmSetKey
- wc_AesGcmDecrypt

Return: 0 On successfully encrypting the input message

Note: When built with WOLFSSL_AFALG_XILINX_AES, the Xilinx AF_ALG kernel interface operates on a combined cipher text + tag buffer, so this function does not honor the exact-size buffer contract described below. Both in and out must be allocated with WC_AES_BLOCK_SIZE (16) bytes of room beyond sz: sz + 16 bytes are sent to the kernel from in (the trailing 16 bytes are scratch space for the tag and their contents are irrelevant), and sz + 16 bytes are read back into out. The tag is additionally copied out to authTag as usual. Both buffers should also be aligned to WOLFSSL_XILINX_ALIGN; an unaligned in is staged through a temporary allocation, or rejected with BAD_ALIGN_E if NO_WOLFSSL_ALLOC_ALIGN is defined.

Example

```
Aes enc;
// initialize Aes structure by calling wc_AesInit() and wc_AesGcmSetKey

byte plain[AES_BLOCK_LENGTH * n]; //n being a positive integer
making plain some multiple of 16 bytes
// initialize plain with msg to encrypt
byte cipher[sizeof(plain)];
byte iv[] = // some 16 byte iv
byte authTag[AUTH_TAG_LENGTH];
byte authIn[] = // Authentication Vector

wc_AesGcmEncrypt(&enc, cipher, plain, sizeof(cipher), iv, sizeof(iv),
        authTag, sizeof(authTag), authIn, sizeof(authIn));
```

```
int wc_AesGcmDecrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    const byte * iv,
    word32 ivSz,
    const byte * authTag,
    word32 authTagSz,
    const byte * authIn,
    word32 authInSz
)
```

This function decrypts the input cipher text, held in the buffer in, and stores the resulting message text in the output buffer out. It also checks the input authentication vector, authIn, against the supplied authentication tag, authTag. If a nonzero error code is returned, the output data is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.

Parameters:

- **aes** pointer to the AES object used to encrypt data
- **out** pointer to the output buffer in which to store the message text size must match in's size (sz)
- **in** pointer to the input buffer holding the cipher text to decrypt size must be a multiple of AES_BLOCK_LENGTH, padded if necessary

- **sz** length of the cipher text to decrypt
- **iv** pointer to the buffer containing the initialization vector
- **ivSz** length of the initialization vector
- **authTag** pointer to the buffer containing the authentication tag
- **authTagSz** length of the desired authentication tag
- **authIn** pointer to the buffer containing the input authentication vector
- **authInSz** length of the input authentication vector

See:

- wc_AesGcmSetKey
- wc_AesGcmEncrypt

Return:

- 0 On successfully decrypting and authenticating the input message
- AES_GCM_AUTH_E If the authentication tag does not match the supplied authentication code vector, authTag.

Note: When built with WOLFSSL_AFALG_XILINX_AES, the Xilinx AF_ALG kernel interface operates on a combined cipher text + tag buffer, so this function does not honor the exact-size buffer contract described below. Both in and out must be allocated with WC_AES_BLOCK_SIZE (16) bytes of room beyond sz. The tag to check against is written into in + sz by this function, which means the in buffer is modified even though it is declared const, and sz + 16 bytes are read back into out. Both buffers should also be aligned to WOLFSSL_XILINX_ALIGN; an unaligned in is staged through a temporary allocation, or rejected with BAD_ALIGN_E if NO_WOLFSSL_ALLOC_ALIGN is defined.

Example

```
Aes enc; //can use the same struct as was passed to wc_AesGcmEncrypt
// initialize aes structure by calling wc_AesInit and wc_AesGcmSetKey
// if not already done

byte cipher[AES_BLOCK_LENGTH * n]; //n being a positive integer
making cipher some multiple of 16 bytes
// initialize cipher with cipher text to decrypt
byte output[sizeof(cipher)];
byte iv[] = // some 16 byte iv
byte authTag[AUTH_TAG_LENGTH];
byte authIn[] = // Authentication Vector

wc_AesGcmDecrypt(&enc, output, cipher, sizeof(cipher), iv, sizeof(iv),
        authTag, sizeof(authTag), authIn, sizeof(authIn));
```

```
int wc_GmacSetKey(
    Gmac * gmac,
    const byte * key,
    word32 len
)
```

This function initializes and sets the key for a GMAC object to be used for Galois Message Authentication.

Parameters:

- **gmac** pointer to the gmac object used for authentication
- **key** 16, 24, or 32 byte secret key for authentication
- **len** length of the key

See:

- wc_GmacUpdate
- wc_AesInit

Return:

- 0 On successfully setting the key
- BAD_FUNC_ARG Returned if key length is invalid.

Example

```
Gmac gmac;
key[] = { some 16, 24, or 32 byte length key };
wc_AesInit(gmac.aes, HEAP_HINT, INVALID_DEVID); // Make sure devId updated
wc_GmacSetKey(&gmac, key, sizeof(key));
```

```
int wc_GmacUpdate(
    Gmac * gmac,
    const byte * iv,
    word32 ivSz,
    const byte * authIn,
    word32 authInSz,
    byte * authTag,
    word32 authTagSz
)
```

This function generates the Gmac hash of the authIn input and stores the result in the authTag buffer. After running wc_GmacUpdate, one should compare the generated authTag to a known authentication tag to verify the authenticity of a message.

Parameters:

- **gmac** pointer to the gmac object used for authentication
- **iv** initialization vector used for the hash
- **ivSz** size of the initialization vector used
- **authIn** pointer to the buffer containing the authentication vector to verify
- **authInSz** size of the authentication vector
- **authTag** pointer to the output buffer in which to store the Gmac hash
- **authTagSz** the size of the output buffer used to store the Gmac hash

See:

- wc_GmacSetKey
- wc_AesInit

Return: 0 On successfully computing the Gmac hash.

Example

```
Gmac gmac;
key[] = { some 16, 24, or 32 byte length key };
iv[] = { some 16 byte length iv };

wc_AesInit(gmac.aes, HEAP_HINT, INVALID_DEVID); // Make sure devId updated
wc_GmacSetKey(&gmac, key, sizeof(key));
authIn[] = { some 16 byte authentication input };
tag[AES_BLOCK_SIZE]; // will store authentication code

wc_GmacUpdate(&gmac, iv, sizeof(iv), authIn, sizeof(authIn), tag,
sizeof(tag));
```

```
int wc_AesCcmSetKey(
    Aes * aes,
    const byte * key,
    word32 keySz
)
```

This function sets the key for an AES object using CCM (Counter with CBC-MAC). It takes a pointer to an AES structure and initializes it with supplied key.

Parameters:

- **aes** aes structure in which to store the supplied key
- **key** 16, 24, or 32 byte secret key for encryption and decryption
- **keySz** size of the supplied key

See:

- wc_AesCcmEncrypt
- wc_AesCcmDecrypt

Return: none

Example

```
Aes enc;
key[] = { some 16, 24, or 32 byte length key };

wc_AesInit(&enc, HEAP_HINT, INVALID_DEVID); // Make sure devId updated
wc_AesCcmSetKey(&enc, key, sizeof(key));
```

```
int wc_AesCcmEncrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * nonce,
    word32 nonceSz,
    byte * authTag,
    word32 authTagSz,
    const byte * authIn,
    word32 authInSz
)
```

This function encrypts the input message, in, into the output buffer, out, using CCM (Counter with CBC-MAC). It subsequently calculates and stores the authorization tag, authTag, from the authIn input.

Parameters:

- **aes** pointer to the AES object used to encrypt data
- **out** pointer to the output buffer in which to store the cipher text
- **in** pointer to the input buffer holding the message to encrypt
- **sz** length of the input message to encrypt
- **nonce** pointer to the buffer containing the nonce (number only used once)
- **nonceSz** length of the nonce
- **authTag** pointer to the buffer in which to store the authentication tag
- **authTagSz** length of the desired authentication tag
- **authIn** pointer to the buffer containing the input authentication vector
- **authInSz** length of the input authentication vector

See:

- wc_AesCcmSetKey
- wc_AesCcmDecrypt

Return: none

Example

```
Aes enc;
// initialize enc with wc_AesInit and wc_AesCcmSetKey

nonce[] = { initialize nonce };
plain[] = { some plain text message };
cipher[sizeof(plain)];

authIn[] = { some 16 byte authentication input };
tag[AES_BLOCK_SIZE]; // will store authentication code

wc_AesCcmEncrypt(&enc, cipher, plain, sizeof(plain), nonce, sizeof(nonce),
        tag, sizeof(tag), authIn, sizeof(authIn));
```

```
int wc_AesCcmDecrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * nonce,
    word32 nonceSz,
    const byte * authTag,
    word32 authTagSz,
    const byte * authIn,
    word32 authInSz
)
```

This function decrypts the input cipher text, in, into the output buffer, out, using CCM (Counter with CBC-MAC). It subsequently calculates the authorization tag, authTag, from the authIn input. If a nonzero error code is returned, the output data is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.

Parameters:

- **aes** pointer to the AES object used to encrypt data
- **out** pointer to the output buffer in which to store the cipher text
- **in** pointer to the input buffer holding the message to encrypt
- **sz** length of the input cipher text to decrypt
- **nonce** pointer to the buffer containing the nonce (number only used once)
- **nonceSz** length of the nonce
- **authTag** pointer to the buffer in which to store the authentication tag
- **authTagSz** length of the desired authentication tag
- **authIn** pointer to the buffer containing the input authentication vector
- **authInSz** length of the input authentication vector

See:

- wc_AesCcmSetKey
- wc_AesCcmEncrypt

Return:

- 0 On successfully decrypting the input message
- AES_CCM_AUTH_E If the authentication tag does not match the supplied authentication code vector, authTag.

Example

```
Aes dec;
// initialize dec with wc_AesInit and wc_AesCcmSetKey

nonce[] = { initialize nonce };
cipher[] = { encrypted message };
plain[sizeof(cipher)];

authIn[] = { some 16 byte authentication input };
```

```
tag[AES_BLOCK_SIZE] = { authentication tag received for verification };

int return = wc_AesCcmDecrypt(&dec, plain, cipher, sizeof(cipher),
nonce, sizeof(nonce),tag, sizeof(tag), authIn, sizeof(authIn));
if(return != 0) {
// decrypt error, invalid authentication code
}
```

```
int wc_AesXtsInit(
    XtsAes * aes,
    void * heap,
    int devId
)
```

This is to initialize an AES-XTS context. It is up to user to call wc_AesXtsFree on aes key when done.

Parameters:

- **aes** AES keys for encrypt/decrypt process
- **heap** heap hint to use for memory. Can be NULL
- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID (-2) if not used

See:

- wc_AesXtsSetKey
- wc_AesXtsSetKeyNoInit
- wc_AesXtsEncrypt
- wc_AesXtsDecrypt
- wc_AesXtsFree

Return: 0 Success

Example

```
XtsAes aes;

if(wc_AesXtsInit(&aes, NULL, INVALID_DEVID) != 0)
{
    // Handle error
}
if(wc_AesXtsSetKeyNoInit(&aes, key, sizeof(key), AES_ENCRYPTION) != 0)
{
    // Handle error
}
wc_AesXtsFree(&aes);
```

```
int wc_AesXtsSetKeyNoInit(
    XtsAes * aes,
    const byte * key,
```

```
    word32 len,
    int dir
)
```

This is to help with setting keys to correct encrypt or decrypt type, after first calling wc_AesXtsInit(). It is up to user to call wc_AesXtsFree on aes key when done.

Parameters:

- **aes** AES keys for encrypt/decrypt process
- **key** buffer holding aes key | tweak key
- **len** length of key buffer in bytes. Should be twice that of key size. i.e. 32 for a 16 byte key.
- **dir** direction, either AES_ENCRYPTION or AES_DECRYPTION

See:

- wc_AesXtsEncrypt
- wc_AesXtsDecrypt
- wc_AesXtsFree

Return: 0 Success

Example

```
XtsAes aes;

if(wc_AesXtsInit(&aes, NULL, 0) != 0)
{
    // Handle error
}
if(wc_AesXtsSetKeyNoInit(&aes, key, sizeof(key), AES_ENCRYPTION, NULL, 0)
   != 0)
{
    // Handle error
}
wc_AesXtsFree(&aes);
```

```
int wc_AesXtsSetKey(
    XtsAes * aes,
    const byte * key,
    word32 len,
    int dir,
    void * heap,
    int devId
)
```

This is to help with setting keys to correct encrypt or decrypt type. It is up to user to call wc_AesXtsFree on aes key when done.

Parameters:

- **aes** AES keys for encrypt/decrypt process
- **key** buffer holding aes key | tweak key
- **len** length of key buffer in bytes. Should be twice that of key size. i.e. 32 for a 16 byte key.
- **dir** direction, either AES_ENCRYPTION or AES_DECRYPTION
- **heap** heap hint to use for memory. Can be NULL
- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID (-2) if not used

See:

- wc_AesXtsInit
- wc_AesXtsSetKeyNoInit
- wc_AesXtsEncrypt
- wc_AesXtsDecrypt
- wc_AesXtsFree

Return: 0 Success

Example

```
XtsAes aes;

if(wc_AesXtsSetKey(&aes, key, sizeof(key), AES_ENCRYPTION, NULL, INVALID_DEVID)
 ↪  != 0)
{
    // Handle error
}
wc_AesXtsFree(&aes);
```

```
int wc_AesXtsEncryptSector(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    word64 sector
)
```

Same process as wc_AesXtsEncrypt but uses a word64 type as the tweak value instead of a byte array. This just converts the word64 to a byte array and calls wc_AesXtsEncrypt.

Parameters:

- **aes** AES keys to use for block encrypt/decrypt
- **out** output buffer to hold cipher text
- **in** input plain text buffer to encrypt
- **sz** size of both out and in buffers
- **sector** value to use for tweak

See:

- wc_AesXtsEncrypt
- wc_AesXtsDecrypt

- wc_AesXtsInit
- wc_AesXtsSetKeyNoInit
- wc_AesXtsSetKey
- wc_AesXtsFree

Return: 0 Success

Example

```
XtsAes aes;
unsigned char plain[SIZE];
unsigned char cipher[SIZE];
word64 s = VALUE;

//set up keys with AES_ENCRYPTION as dir

if(wc_AesXtsEncryptSector(&aes, cipher, plain, SIZE, s) != 0)
{
    // Handle error
}
wc_AesXtsFree(&aes);
```

```
int wc_AesXtsDecryptSector(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    word64 sector
)
```

Same process as wc_AesXtsDecrypt but uses a word64 type as the tweak value instead of a byte array. This just converts the word64 to a byte array.

Parameters:

- **aes** AES keys to use for block encrypt/decrypt
- **out** output buffer to hold plain text
- **in** input cipher text buffer to decrypt
- **sz** size of both out and in buffers
- **sector** value to use for tweak

See:

- wc_AesXtsEncrypt
- wc_AesXtsDecrypt
- wc_AesXtsInit
- wc_AesXtsSetKeyNoInit
- wc_AesXtsSetKey
- wc_AesXtsFree

Return: 0 Success

Example

```
XtsAes aes;
unsigned char plain[SIZE];
unsigned char cipher[SIZE];
word64 s = VALUE;

//set up aes key with AES_DECRYPTION as dir and tweak with AES_ENCRYPTION

if(wc_AesXtsDecryptSector(&aes, plain, cipher, SIZE, s) != 0)
{
    // Handle error
}
wc_AesXtsFree(&aes);
```

```
int wc_AesXtsEncrypt(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    const byte * i,
    word32 iSz
)
```

AES with XTS mode. (XTS) XEX encryption with Tweak and cipher text Stealing.

Parameters:

- **aes** AES keys to use for block encrypt/decrypt
- **out** output buffer to hold cipher text
- **in** input plain text buffer to encrypt
- **sz** size of both out and in buffers
- **i** value to use for tweak
- **iSz** size of i buffer, should always be AES_BLOCK_SIZE but having this input adds a sanity check on how the user calls the function.

See:

- wc_AesXtsDecrypt
- wc_AesXtsInit
- wc_AesXtsSetKeyNoInit
- wc_AesXtsSetKey
- wc_AesXtsFree

Return: 0 Success

Example

```
XtsAes aes;
unsigned char plain[SIZE];
```

```
unsigned char cipher[SIZE];
unsigned char i[AES_BLOCK_SIZE];

//set up key with AES_ENCRYPTION as dir

if(wc_AesXtsEncrypt(&aes, cipher, plain, SIZE, i, sizeof(i)) != 0)
{
    // Handle error
}
wc_AesXtsFree(&aes);
```

```
int wc_AesXtsDecrypt(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    const byte * i,
    word32 iSz
)
```

Same process as encryption but Aes key is AES_DECRYPTION type.

Parameters:

- **aes** AES keys to use for block encrypt/decrypt
- **out** output buffer to hold plain text
- **in** input cipher text buffer to decrypt
- **sz** size of both out and in buffers
- **i** value to use for tweak
- **iSz** size of i buffer, should always be AES_BLOCK_SIZE but having this input adds a sanity check on how the user calls the function.

See:

- wc_AesXtsEncrypt
- wc_AesXtsInit
- wc_AesXtsSetKeyNoInit
- wc_AesXtsSetKey
- wc_AesXtsFree

Return: 0 Success

Example

```
XtsAes aes;
unsigned char plain[SIZE];
unsigned char cipher[SIZE];
unsigned char i[AES_BLOCK_SIZE];

//set up key with AES_DECRYPTION as dir and tweak with AES_ENCRYPTION
```

```
if(wc_AesXtsDecrypt(&aes, plain, cipher, SIZE, i, sizeof(i)) != 0)
{
    // Handle error
}
wc_AesXtsFree(&aes);
```

```
int wc_AesXtsFree(
    XtsAes * aes
)
```

This is to free up any resources used by the XtsAes structure.

Parameters:

- **aes** AES keys to free

See:

- wc_AesXtsEncrypt
- wc_AesXtsDecrypt
- wc_AesXtsInit
- wc_AesXtsSetKeyNoInit
- wc_AesXtsSetKey

Return: 0 Success

Example

```
XtsAes aes;

if(wc_AesXtsSetKey(&aes, key, sizeof(key), AES_ENCRYPTION, NULL, 0) != 0)
{
    // Handle error
}
wc_AesXtsFree(&aes);
```

```
int wc_AesInit(
    Aes * aes,
    void * heap,
    int devId
)
```

Initialize Aes structure. Sets heap hint to be used and ID for use with async hardware. It is up to the user to call wc_AesFree on the Aes structure when done.

Parameters:

- **aes** aes structure in to initialize
- **heap** heap hint to use for malloc / free if needed

- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID (-2) if not used

See:

- wc_AesSetKey
- wc_AesSetIV
- wc_AesFree

Return: 0 Success

Example

```
Aes enc;
void* hint = NULL;
int devId = INVALID_DEVID; //if not using async INVALID_DEVID is default

//heap hint could be set here if used

wc_AesInit(&enc, hint, devId);
```

```
void wc_AesFree(
    Aes * aes
)
```

free resources associated with the Aes structure when applicable. Internally may sometimes be a no-op but still recommended to call in all cases as a general best-practice (IE if application code is ported for use on new environments where the call is applicable).

Parameters:

- **aes** aes structure in to free

See: wc_AesInit

Return: no return (void function)

Example

```
Aes enc;
void* hint = NULL;
int devId = INVALID_DEVID; //if not using async INVALID_DEVID is default

//heap hint could be set here if used

wc_AesInit(&enc, hint, devId);
// ... do some interesting things ...
wc_AesFree(&enc);
```

```
int wc_AesCfbEncrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

AES with CFB mode.

Parameters:

- **aes** AES keys to use for block encrypt/decrypt
- **out** output buffer to hold cipher text must be at least as large as inputbuffer)
- **in** input plain text buffer to encrypt
- **sz** size of input buffer

See:

- wc_AesCfbDecrypt
- wc_AesSetKey

Return: 0 Success and negative error values on failure

Example

```
Aes aes;
unsigned char plain[SIZE];
unsigned char cipher[SIZE];

//set up key with AES_ENCRYPTION as dir for both encrypt and decrypt

if(wc_AesCfbEncrypt(&aes, cipher, plain, SIZE) != 0)
{
    // Handle error
}
```

```
int wc_AesCfbDecrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

AES with CFB mode.

Parameters:

- **aes** AES keys to use for block encrypt/decrypt
- **out** output buffer to hold decrypted text must be at least as large as inputbuffer)
- **in** input buffer to decrypt
- **sz** size of input buffer

See:

- wc_AesCfbEncrypt
- wc_AesSetKey

Return: 0 Success and negative error values on failure

Example

```
Aes aes;
unsigned char plain[SIZE];
unsigned char cipher[SIZE];

//set up key with AES_ENCRYPTION as dir for both encrypt and decrypt

if(wc_AesCfbDecrypt(&aes, plain, cipher, SIZE) != 0)
{
    // Handle error
}
```

```
int wc_AesSivEncrypt(
    const byte * key,
    word32 keySz,
    const byte * assoc,
    word32 assocSz,
    const byte * nonce,
    word32 nonceSz,
    const byte * in,
    word32 inSz,
    byte * siv,
    byte * out
)
```

This function performs SIV (synthetic initialization vector) encryption as described in RFC 5297.

Parameters:

- **key** Byte buffer containing the key to use.
- **keySz** Length of the key buffer in bytes.
- **assoc** Additional, authenticated associated data (AD).
- **assocSz** Length of AD buffer in bytes.
- **nonce** A number used once. Used by the algorithm in the same manner as the AD.
- **nonceSz** Length of nonce buffer in bytes.
- **in** Plaintext buffer to encrypt.
- **inSz** Length of plaintext buffer.
- **siv** The SIV output by S2V (see RFC 5297 2.4).
- **out** Buffer to hold the ciphertext. Should be the same length as the plaintext buffer.

See: wc_AesSivDecrypt

Return:

- 0 On successful encryption.
- BAD_FUNC_ARG If key, SIV, or output buffer are NULL. Also returned if the key size isn't 32, 48, or 64 bytes.
- Other Other negative error values returned if AES or CMAC operations fail.

Example

```
byte key[] = { some 32, 48, or 64 byte key };
byte assoc[] = {0x01, 0x2, 0x3};
byte nonce[] = {0x04, 0x5, 0x6};
byte plainText[] = {0xDE, 0xAD, 0xBE, 0xEF};
byte siv[AES_BLOCK_SIZE];
byte cipherText[sizeof(plainText)];
if (wc_AesSivEncrypt(key, sizeof(key), assoc, sizeof(assoc), nonce,
    sizeof(nonce), plainText, sizeof(plainText), siv, cipherText) != 0) {
    // failed to encrypt
}
```

```
int wc_AesSivDecrypt(
    const byte * key,
    word32 keySz,
    const byte * assoc,
    word32 assocSz,
    const byte * nonce,
    word32 nonceSz,
    const byte * in,
    word32 inSz,
    byte * siv,
    byte * out
)
```

This function performs SIV (synthetic initialization vector) decryption as described in RFC 5297. If a nonzero error code is returned, the output data is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.

Parameters:

- **key** Byte buffer containing the key to use.
- **keySz** Length of the key buffer in bytes.
- **assoc** Additional, authenticated associated data (AD).
- **assocSz** Length of AD buffer in bytes.
- **nonce** A number used once. Used by the underlying algorithm in the same manner as the AD.
- **nonceSz** Length of nonce buffer in bytes.
- **in** Ciphertext buffer to decrypt.
- **inSz** Length of ciphertext buffer.
- **siv** The SIV that accompanies the ciphertext (see RFC 5297 2.4).
- **out** Buffer to hold the decrypted plaintext. Should be the same length as the ciphertext buffer.

See: wc_AesSivEncrypt

Return:

- 0 On successful decryption.
- BAD_FUNC_ARG If key, SIV, or output buffer are NULL. Also returned if the key size isn't 32, 48, or 64 bytes.
- AES_SIV_AUTH_E If the SIV derived by S2V doesn't match the input SIV (see RFC 5297 2.7).
- Other Other negative error values returned if AES or CMAC operations fail.

Example

```
byte key[] = { some 32, 48, or 64 byte key };
byte assoc[] = {0x01, 0x2, 0x3};
byte nonce[] = {0x04, 0x5, 0x6};
byte cipherText[] = {0xDE, 0xAD, 0xBE, 0xEF};
byte siv[AES_BLOCK_SIZE] = { the SIV that came with the ciphertext };
byte plainText[sizeof(cipherText)];
if (wc_AesSivDecrypt(key, sizeof(key), assoc, sizeof(assoc), nonce,
    sizeof(nonce), cipherText, sizeof(cipherText), siv, plainText) != 0) {
    // failed to decrypt
}
```

```
WOLFSSL_API int wc_AesEaxEncryptAuth(
    const byte * key,
    word32 keySz,
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * nonce,
    word32 nonceSz,
    byte * authTag,
    word32 authTagSz,
    const byte * authIn,
    word32 authInSz
)
```

This function performs AES EAX encryption and authentication as described in "EAX: A Conventional Authenticated-Encryption Mode" (https://eprint.iacr.org/2003/069). It is a "one-shot" API that performs all encryption and authentication operations in one function call.

Parameters:

- **key** buffer containing the key to use
- **keySz** length of the key buffer in bytes
- **out** buffer to hold the ciphertext. Should be the same length as the plaintext buffer
- **in** plaintext buffer to encrypt
- **inSz** length of plaintext buffer
- **nonce** the cryptographic nonce to use for EAX operations
- **nonceSz** length of nonce buffer in bytes
- **authTag** pointer to the buffer in which to store the authentication tag
- **authTagSz** length of the desired authentication tag
- **authIn** pointer to the buffer containing input data to authenticate
- **authInSz** length of the input authentication data

See: wc_AesEaxDecryptAuth

Return:

- 0 on successful encryption.
- BAD_FUNC_ARG if input or output buffers are NULL. Also returned if the key size isn't a valid AES key size (16, 24, or 32 bytes)
- other negative error values returned if AES or CMAC operations fail.

Example

```
byte key[] = { some 32, 48, or 64 byte key };
byte nonce[] = {0x04, 0x5, 0x6};
byte plainText[] = {0xDE, 0xAD, 0xBE, 0xEF};
byte authIn[] = {0x01, 0x2, 0x3};

byte cipherText[sizeof(plainText)]; // output ciphertext
byte authTag[length, up to AES_BLOCK_SIZE]; // output authTag

if (wc_AesEaxEncrypt(key, sizeof(key),
                     cipherText, plainText, sizeof(plainText),
                     nonce, sizeof(nonce),
                     authTag, sizeof(authTag),
                     authIn, sizeof(authIn)) != 0) {
    // failed to encrypt
}
```

```
WOLFSSL_API int wc_AesEaxDecryptAuth(
    const byte * key,
    word32 keySz,
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * nonce,
    word32 nonceSz,
    const byte * authTag,
    word32 authTagSz,
    const byte * authIn,
    word32 authInSz
)
```

This function performs AES EAX decryption and authentication as described in "EAX: A Conventional Authenticated-Encryption Mode" (https://eprint.iacr.org/2003/069). It is a "one-shot" API that performs all decryption and authentication operations in one function call. If a nonzero error code is returned, the output data is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.

Parameters:

- **key** byte buffer containing the key to use
- **keySz** length of the key buffer in bytes

- **out** buffer to hold the plaintext. Should be the same length as the input ciphertext buffer
- **in** ciphertext buffer to decrypt
- **inSz** length of ciphertext buffer
- **nonce** the cryptographic nonce to use for EAX operations
- **nonceSz** length of nonce buffer in bytes
- **authTag** buffer that holds the authentication tag to check the authenticity of the data against
- **authTagSz** Length of the input authentication tag
- **authIn** pointer to the buffer containing input data to authenticate
- **authInSz** length of the input authentication data

See: wc_AesEaxEncryptAuth

Return:

- 0 on successful decryption
- BAD_FUNC_ARG if input or output buffers are NULL. Also returned if the key size isn't a valid AES key size (16, 24, or 32 bytes)
- AES_EAX_AUTH_E If the authentication tag does not match the supplied authentication code vector authTag
- other negative error values returned if AES or CMAC operations fail.

Example

```
byte key[] = { some 32, 48, or 64 byte key };
byte nonce[] = {0x04, 0x5, 0x6};
byte cipherText[] = {0xDE, 0xAD, 0xBE, 0xEF};
byte authIn[] = {0x01, 0x2, 0x3};

byte plainText[sizeof(cipherText)]; // output plaintext
byte authTag[length, up to AES_BLOCK_SIZE]; // output authTag

if (wc_AesEaxDecrypt(key, sizeof(key),
                     cipherText, plainText, sizeof(plainText),
                     nonce, sizeof(nonce),
                     authTag, sizeof(authTag),
                     authIn, sizeof(authIn)) != 0) {
    // failed to encrypt
}
```

```
WOLFSSL_API int wc_AesEaxInit(
    AesEax * eax,
    const byte * key,
    word32 keySz,
    const byte * nonce,
    word32 nonceSz,
    const byte * authIn,
    word32 authInSz
)
```

This function initializes an AesEax object for use in authenticated encryption or decryption. This function must be called on an AesEax object before using it with any of the AES EAX incremental API functions. It does not need to be called if using the one-shot EAX API functions. All AesEax instances

initialized with this function need to be freed with a call to wc_AesEaxFree() when done using the instance.

Parameters:

- **eax** AES EAX structure holding the context of the AEAD operation
- **key** 16, 24, or 32 byte secret key for encryption and decryption
- **keySz** length of the supplied key in bytes
- **nonce** the cryptographic nonce to use for EAX operations
- **nonceSz** length of nonce buffer in bytes
- **authIn** (optional) input data to add to the authentication stream This argument should be NULL if not used
- **authInSz** size in bytes of the input authentication data

See:

- wc_AesEaxEncryptUpdate
- wc_AesEaxDecryptUpdate
- wc_AesEaxAuthDataUpdate
- wc_AesEaxEncryptFinal
- wc_AesEaxDecryptFinal
- wc_AesEaxFree

Return:

- 0 on success
- error code on failure

Example

```
AesEax eax;
key[]   = { some 16, 24, or 32 byte length key };
nonce[] = { some arbitrary length nonce };
authIn[] = { some data to add to the authentication stream };
plainText[] = {some plaintext data to encrypt};

cipherText[sizeof(plainText)]; // buffer to hold cipherText
authTag[length, up to AES_BLOCK_SIZE]; // buffer to hold computed auth data

AesEax eax;

if ((ret = wc_AesEaxInit(eax,
                         key, keySz,
                         nonce, nonceSz,
                         authIn, authInSz)) != 0) {
    goto cleanup;
}

// if we wanted to add more auth data, we could provide it at this point,
// otherwise we use NULL for the authIn parameter, with authIn size of 0
if ((ret = wc_AesEaxEncryptUpdate(eax,
                                  cipherText, plainText, sizeof(plainText),
```

```
                                    NULL, 0)) != 0) {
    goto cleanup;
}

if ((ret = wc_AesEaxEncryptFinal(eax, authTag, sizeof(authTag))) != 0) {
    goto cleanup;
}

cleanup:
    wc_AesEaxFree(eax);



WOLFSSL_API int wc_AesEaxEncryptUpdate(
    AesEax * eax,
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * authIn,
    word32 authInSz
)
```

This function uses AES EAX to encrypt input data, and optionally, add more input data to the authentication stream. eax must have been previously initialized with a call to wc_AesEaxInit.

Parameters:

- **eax** AES EAX structure holding the context of the AEAD operation
- **out** output buffer holding the ciphertext
- **in** input buffer holding the plaintext to encrypt
- **inSz** size in bytes of the input data buffer
- **authIn** (optional) input data to add to the authentication stream This argument should be NULL if not used
- **authInSz** size in bytes of the input authentication data

See:

- wc_AesEaxInit
- wc_AesEaxDecryptUpdate
- wc_AesEaxAuthDataUpdate
- wc_AesEaxEncryptFinal
- wc_AesEaxDecryptFinal
- wc_AesEaxFree

Return:

- 0 on success
- error code on failure

Example

```
AesEax eax;
key[]   = { some 16, 24, or 32 byte length key };
nonce[] = { some arbitrary length nonce };
authIn[] = { some data to add to the authentication stream };
plainText[] = {some plaintext data to encrypt};

cipherText[sizeof(plainText)]; // buffer to hold cipherText
authTag[length, up to AES_BLOCK_SIZE]; // buffer to hold computed auth data

AesEax eax;

if ((ret = wc_AesEaxInit(eax,
                         key, keySz,
                         nonce, nonceSz,
                         authIn, authInSz)) != 0) {
    goto cleanup;
}

// if we wanted to add more auth data, we could provide it at this point,
// otherwise we use NULL for the authIn parameter, with authInSz of 0
if ((ret = wc_AesEaxEncryptUpdate(eax,
                                  cipherText, plainText, sizeof(plainText),
                                  NULL, 0)) != 0) {
    goto cleanup;
}

if ((ret = wc_AesEaxEncryptFinal(eax, authTag, sizeof(authTag))) != 0) {
    goto cleanup;
}

cleanup:
    wc_AesEaxFree(eax);



WOLFSSL_API int wc_AesEaxDecryptUpdate(
    AesEax * eax,
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * authIn,
    word32 authInSz
)
```

This function uses AES EAX to decrypt input data, and optionally, add more input data to the authentication stream. eax must have been previously initialized with a call to wc_AesEaxInit.

Parameters:

- **eax** AES EAX structure holding the context of the AEAD operation
- **out** output buffer holding the decrypted plaintext
- **in** input buffer holding the ciphertext
- **inSz** size in bytes of the input data buffer

- **authIn** (optional) input data to add to the authentication stream This argument should be NULL if not used
- **authInSz** size in bytes of the input authentication data

See:

- wc_AesEaxInit
- wc_AesEaxEncryptUpdate
- wc_AesEaxAuthDataUpdate
- wc_AesEaxEncryptFinal
- wc_AesEaxDecryptFinal
- wc_AesEaxFree

Return:

- 0 on success
- error code on failure

Example

```
AesEax eax;
key[]   = { some 16, 24, or 32 byte length key };
nonce[] = { some arbitrary length nonce };
authIn[] = { some data to add to the authentication stream };
cipherText[] = {some encrypted data};

plainText[sizeof(cipherText)]; // buffer to hold decrypted data
// auth tag is generated elsewhere by the encrypt AEAD operation
authTag[length, up to AES_BLOCK_SIZE] = { the auth tag };

AesEax eax;

if ((ret = wc_AesEaxInit(eax,
                         key, keySz,
                         nonce, nonceSz,
                         authIn, authInSz)) != 0) {
    goto cleanup;
}

// if we wanted to add more auth data, we could provide it at this point,
// otherwise we use NULL for the authIn parameter, with authInSz of 0
if ((ret = wc_AesEaxDecryptUpdate(eax,
                                  plainText, cipherText, sizeof(cipherText),
                                  NULL, 0)) != 0) {
    goto cleanup;
}

if ((ret = wc_AesEaxDecryptFinal(eax, authTag, sizeof(authTag))) != 0) {
    goto cleanup;
}

cleanup:
    wc_AesEaxFree(eax);
```

```
WOLFSSL_API int wc_AesEaxAuthDataUpdate(
    AesEax * eax,
    const byte * authIn,
    word32 authInSz
)
```

This function adds input data to the authentication stream. eax must have been previously initialized with a call to wc_AesEaxInit.

Parameters:

- **eax** AES EAX structure holding the context of the AEAD operation
- **authIn** input data to add to the authentication stream
- **authInSz** size in bytes of the input authentication data

See:

- wc_AesEaxInit
- wc_AesEaxEncryptUpdate
- wc_AesEaxDecryptUpdate
- wc_AesEaxEncryptFinal
- wc_AesEaxDecryptFinal
- wc_AesEaxFree

Return:

- 0 on success
- error code on failure

Example

```
AesEax eax;
key[]   = { some 16, 24, or 32 byte length key };
nonce[] = { some arbitrary length nonce };
authIn[] = { some data to add to the authentication stream };
cipherText[] = {some encrypted data};

plainText[sizeof(cipherText)]; // buffer to hold decrypted data
// auth tag is generated elsewhere by the encrypt AEAD operation
authTag[length, up to AES_BLOCK_SIZE] = { the auth tag };

AesEax eax;

// No auth data to add here
if ((ret = wc_AesEaxInit(eax,
                         key, keySz,
                         nonce, nonceSz,
                         NULL, 0)) != 0) {
    goto cleanup;
}
```

```
// No auth data to add here, added later with wc_AesEaxAuthDataUpdate
if ((ret = wc_AesEaxDecryptUpdate(eax,
                                  plainText, cipherText, sizeof(cipherText),
                                  NULL, 0)) != 0) {
    goto cleanup;
}

if ((ret = wc_AesEaxAuthDataUpdate(eax, authIn, sizeof(authIn))) != 0) {
    goto cleanup;
}

if ((ret = wc_AesEaxDecryptFinal(eax, authTag, sizeof(authTag))) != 0) {
    goto cleanup;
}

cleanup:
    wc_AesEaxFree(eax);
```

```
WOLFSSL_API int wc_AesEaxEncryptFinal(
    AesEax * eax,
    byte * authTag,
    word32 authTagSz
)
```

This function finalizes the encrypt AEAD operation, producing an auth tag over the current authentication stream. eax must have been previously initialized with a call to wc_AesEaxInit. When done using the AesEax context structure, make sure to free it using wc_AesEaxFree.

Parameters:

- **eax** AES EAX structure holding the context of the AEAD operation
- **authTag[out]** buffer that will hold the computed auth tag
- **authTagSz** size in bytes of authTag

See:

- wc_AesEaxInit
- wc_AesEaxEncryptUpdate
- wc_AesEaxDecryptUpdate
- wc_AesEaxAuthDataUpdate
- wc_AesEaxDecryptFinal
- wc_AesEaxFree

Return:

- 0 on success
- error code on failure

Example

```
AesEax eax;
key[]   = { some 16, 24, or 32 byte length key };
nonce[] = { some arbitrary length nonce };
authIn[] = { some data to add to the authentication stream };
plainText[] = {some plaintext data to encrypt};

cipherText[sizeof(plainText)]; // buffer to hold cipherText
authTag[length, up to AES_BLOCK_SIZE]; // buffer to hold computed auth data

AesEax eax;

if ((ret = wc_AesEaxInit(eax,
                         key, keySz,
                         nonce, nonceSz,
                         authIn, authInSz)) != 0) {
    goto cleanup;
}

// if we wanted to add more auth data, we could provide it at this point,
// otherwise we use NULL for the authIn parameter, with authInSz of 0
if ((ret = wc_AesEaxEncryptUpdate(eax,
                                  cipherText, plainText, sizeof(plainText),
                                  NULL, 0)) != 0) {
    goto cleanup;
}

if ((ret = wc_AesEaxEncryptFinal(eax, authTag, sizeof(authTag))) != 0) {
    goto cleanup;
}

cleanup:
    wc_AesEaxFree(eax);



WOLFSSL_API int wc_AesEaxDecryptFinal(
    AesEax * eax,
    const byte * authIn,
    word32 authInSz
)
```

This function finalizes the decrypt AEAD operation, finalizing the auth tag computation and checking it for validity against the user supplied tag. eax must have been previously initialized with a call to wc_AesEaxInit. When done using the AesEax context structure, make sure to free it using wc_AesEaxFree.

Parameters:

- **eax** AES EAX structure holding the context of the AEAD operation
- **authIn** input auth tag to check computed auth tag against
- **authInSz** size in bytes of authIn

See:

- wc_AesEaxInit
- wc_AesEaxEncryptUpdate
- wc_AesEaxDecryptUpdate
- wc_AesEaxAuthDataUpdate
- wc_AesEaxEncryptFinal
- wc_AesEaxFree

Return:

- 0 if data is authenticated successfully
- AES_EAX_AUTH_E if the authentication tag does not match the supplied authentication code vector authIn
- other error code on failure

Example

```
AesEax eax;
key[]   = { some 16, 24, or 32 byte length key };
nonce[] = { some arbitrary length nonce };
authIn[] = { some data to add to the authentication stream };
cipherText[] = {some encrypted data};

plainText[sizeof(cipherText)]; // buffer to hold decrypted data
// auth tag is generated elsewhere by the encrypt AEAD operation
authTag[length, up to AES_BLOCK_SIZE] = { the auth tag };

AesEax eax;

if ((ret = wc_AesEaxInit(eax,
                         key, keySz,
                         nonce, nonceSz,
                         authIn, authInSz)) != 0) {
    goto cleanup;
}

// if we wanted to add more auth data, we could provide it at this point,
// otherwise we use NULL for the authIn parameter, with authInSz of 0
if ((ret = wc_AesEaxDecryptUpdate(eax,
                                  plainText, cipherText, sizeof(cipherText),
                                  NULL, 0)) != 0) {
    goto cleanup;
}

if ((ret = wc_AesEaxDecryptFinal(eax, authTag, sizeof(authTag))) != 0) {
    goto cleanup;
}

cleanup:
    wc_AesEaxFree(eax);
```

```
WOLFSSL_API int wc_AesEaxFree(
```

```
    AesEax * eax
)
```

This frees up any resources, specifically keys, used by the Aes instance inside the AesEax wrapper struct. It should be called on the AesEax struct after it has been initialized with wc_AesEaxInit, and all desired EAX operations are complete.

Parameters:

- **eaxAES** EAX instance to free

See:

- wc_AesEaxInit
- wc_AesEaxEncryptUpdate
- wc_AesEaxDecryptUpdate
- wc_AesEaxAuthDataUpdate
- wc_AesEaxEncryptFinal
- wc_AesEaxDecryptFinal

Return: 0 Success

Example

```
AesEax eax;

if(wc_AesEaxInit(eax, key, keySz, nonce, nonceSz, authIn, authInSz) != 0) {
    // handle errors, then free
    wc_AesEaxFree(&eax);
}
```

```
int wc_AesCtsEncrypt(
    const byte * key,
    word32 keySz,
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * iv
)
```

This function performs AES encryption using Ciphertext Stealing (CTS) mode. It is a one-shot API that handles all operations in a single call.

Parameters:

- **key** pointer to the AES key used for encryption.
- **keySz** size of the AES key in bytes (16, 24, or 32 bytes).
- **out** buffer to hold the encrypted ciphertext. Must be at least the size of the input.
- **in** pointer to the plaintext input data to encrypt.
- **inSz** size of the plaintext input data in bytes.
- **iv** pointer to the initialization vector (IV) used for encryption. Must be 16 bytes.

- **key** pointer to the AES key used for encryption.
- **keySz** size of the AES key in bytes (16, 24, or 32 bytes).
- **out** buffer to hold the encrypted ciphertext. Must be at least the same size as the input plaintext.
- **in** pointer to the plaintext input data to encrypt.
- **inSz** size of the plaintext input data in bytes.
- **iv** pointer to the initialization vector (IV) used for encryption. Must be 16 bytes. *Example*

```
byte key[16] = { 0 };
byte iv[16] = { 0 };
byte plaintext[] = { 0x01, 0x02, 0x03, 0x04, 0x05 };
byte ciphertext[sizeof(plaintext)];
int ret = wc_AesCtsEncrypt(key, sizeof(key), ciphertext, plaintext,
                           sizeof(plaintext), iv);
if (ret != 0) {
    // handle encryption error
}
```

See:

- wc_AesCtsDecrypt
- wc_AesCtsDecrypt

Return:

- 0 on successful encryption.
- BAD_FUNC_ARG if input arguments are invalid.
- other negative error codes for encryption failures.
- 0 on successful encryption.
- BAD_FUNC_ARG if input arguments are invalid.
- other negative error codes for encryption failures.

Example

```
    byte key[16] = { 0 };
    byte iv[16] = { 0 };
    byte plaintext[] = { 0x01, 0x02, 0x03, 0x04, 0x05 };
    byte ciphertext[sizeof(plaintext)];

    int ret = wc_AesCtsEncrypt(key, sizeof(key), ciphertext, plaintext,
        sizeof(plaintext), iv);
    if (ret != 0) {
    // handle encryption error
}
```

```
int wc_AesCtsDecrypt(
    const byte * key,
    word32 keySz,
    byte * out,
    const byte * in,
    word32 inSz,
```

```
    const byte * iv
)
```

This function performs AES decryption using Ciphertext Stealing (CTS) mode. It is a one-shot API that handles all operations in a single call.

Parameters:

- **key** pointer to the AES key used for decryption.
- **keySz** size of the AES key in bytes (16, 24, or 32 bytes).
- **out** buffer to hold the decrypted plaintext. Must be at least the same size as the input ciphertext.
- **in** pointer to the ciphertext input data to decrypt.
- **inSz** size of the ciphertext input data in bytes.
- **iv** pointer to the initialization vector (IV) used for decryption. Must be 16 bytes. *Example*

```
byte key[16] = { 0 };
byte iv[16] = { 0 };
byte ciphertext[] = { 0x01, 0x02, 0x03, 0x04, 0x05 };
byte plaintext[sizeof(ciphertext)];
int ret = wc_AesCtsDecrypt(key, sizeof(key), plaintext, ciphertext,
                           sizeof(ciphertext), iv);
if (ret != 0) {
    // handle decryption error
}
```

See: wc_AesCtsEncrypt

Return:

- 0 on successful decryption.
- BAD_FUNC_ARG if input arguments are invalid.
- other negative error codes for decryption failures.

```
int wc_AesCtsEncryptUpdate(
    Aes * aes,
    byte * out,
    word32 * outSz,
    const byte * in,
    word32 inSz
)
```

This function performs an update step of the AES CTS encryption. It processes a chunk of plaintext and stores intermediate data.

Parameters:

- **aes** pointer to the Aes structure holding the context of the operation.
- **out** buffer to hold the encrypted ciphertext. Must be large enough to store the output from this update step.

- **outSz** size in bytes of the output data written to the out buffer. On input, it should contain the maximum number of bytes that can be written to the out buffer.
- **in** pointer to the plaintext input data to encrypt.
- **inSz** size of the plaintext input data in bytes. *Example*

```
Aes aes;
wc_AesInit(&aes, NULL, INVALID_DEVID);
byte key[16] = { 0 };
byte iv[16] = { 0 };
byte plaintext[] = { ... };
byte ciphertext[sizeof(plaintext)];
word32 outSz = sizeof(ciphertext);
wc_AesSetKey(&aes, key, sizeof(key), iv, AES_ENCRYPTION);
int ret = wc_AesCtsEncryptUpdate(&aes, ciphertext, &outSz, plaintext,
 ↪  sizeof(plaintext));
if (ret != 0) {
    // handle error
}
wc_AesFree(&aes);
```

See: wc_AesCtsDecryptUpdate

Return:

- 0 on successful processing.
- BAD_FUNC_ARG if input arguments are invalid.

```
int wc_AesCtsEncryptFinal(
    Aes * aes,
    byte * out,
    word32 * outSz
)
```

This function finalizes the AES CTS encryption operation. It processes any remaining plaintext and completes the encryption.

Parameters:

- **aes** pointer to the Aes structure holding the context of the operation.
- **out** buffer to hold the final encrypted ciphertext. Must be large enough to store any remaining ciphertext from this final step.
- **outSz** size in bytes of the output data written to the out buffer. On input, it should contain the maximum number of bytes that can be written to the out buffer. *Example*

```
Aes aes;
wc_AesInit(&aes, NULL, INVALID_DEVID);
byte key[16] = { 0 };
byte iv[16] = { 0 };
byte plaintext[] = { ... };
byte ciphertext[sizeof(plaintext)];
```

```
word32 outSz = sizeof(ciphertext);
wc_AesSetKey(&aes, key, sizeof(key), iv, AES_ENCRYPTION);
// Perform any required update steps using wc_AesCtsEncryptUpdate
int ret = wc_AesCtsEncryptFinal(&aes, ciphertext, &outSz);
if (ret != 0) {
    // handle error
}
wc_AesFree(&aes);
```

See: wc_AesCtsDecryptFinal

Return:

- 0 on successful encryption completion.
- BAD_FUNC_ARG if input arguments are invalid.

```
int wc_AesCtsDecryptUpdate(
    Aes * aes,
    byte * out,
    word32 * outSz,
    const byte * in,
    word32 inSz
)
```

This function performs an update step of the AES CTS decryption. It processes a chunk of ciphertext and stores intermediate data.

Parameters:

- **aes** pointer to the Aes structure holding the context of the operation.
- **out** buffer to hold the decrypted plaintext. Must be large enough to store the output from this update step.
- **outSz** size in bytes of the output data written to the out buffer. On input, it should contain the maximum number of bytes that can be written to the out buffer.
- **in** pointer to the ciphertext input data to decrypt.
- **inSz** size of the ciphertext input data in bytes. *Example*

```
Aes aes;
wc_AesInit(&aes, NULL, INVALID_DEVID);
byte key[16] = { 0 };
byte iv[16] = { 0 };
byte ciphertext[] = { ... };
byte plaintext[sizeof(ciphertext)];
word32 outSz = sizeof(plaintext);
wc_AesSetKey(&aes, key, sizeof(key), iv, AES_DECRYPTION);
int ret = wc_AesCtsDecryptUpdate(&aes, plaintext, &outSz, ciphertext,
 ↪  sizeof(ciphertext));
if (ret != 0) {
    // handle error
}
wc_AesFree(&aes);
```

See: wc_AesCtsEncryptUpdate

Return:

- 0 on successful processing.
- BAD_FUNC_ARG if input arguments are invalid.

```
int wc_AesCtsDecryptFinal(
    Aes * aes,
    byte * out,
    word32 * outSz
)
```

This function finalizes the AES CTS decryption operation. It processes any remaining ciphertext and completes the decryption.

Parameters:

- **aes** pointer to the Aes structure holding the context of the operation.
- **out** buffer to hold the final decrypted plaintext. Must be large enough to store any remaining plaintext from this final step.
- **outSz** size in bytes of the output data written to the out buffer. On input, it should contain the maximum number of bytes that can be written to the out buffer. *Example*

```
Aes aes;
wc_AesInit(&aes, NULL, INVALID_DEVID);
byte key[16] = { 0 };
byte iv[16] = { 0 };
byte ciphertext[] = { ... };
byte plaintext[sizeof(ciphertext)];
word32 outSz = sizeof(plaintext);
wc_AesSetKey(&aes, key, sizeof(key), iv, AES_DECRYPTION);
// Perform any required update steps using wc_AesCtsDecryptUpdate
int ret = wc_AesCtsDecryptFinal(&aes, plaintext, &outSz);
if (ret != 0) {
    // handle error
}
wc_AesFree(&aes);
```

See: wc_AesCtsEncryptFinal

Return:

- 0 on successful decryption completion.
- BAD_FUNC_ARG if input arguments are invalid.

```
int wc_AesCfb1Encrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function encrypts data using AES CFB-1 mode (1-bit feedback). It processes data one bit at a time, making it suitable for bit-oriented applications.

Parameters:

- **aes** pointer to the AES structure containing the key
- **out** pointer to the output buffer to store encrypted data
- **in** pointer to the input buffer containing data to encrypt (packed to left, e.g., 101 is 0x90)
- **sz** size of input in bits

See:

- wc_AesCfb1Decrypt
- wc_AesCfb8Encrypt

Return:

- 0 On success.
- BAD_FUNC_ARG If aes, out, or in is NULL.
- Other negative values on error.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte iv[16] = { }; // initialization vector
byte plaintext[1] = { 0x90 }; // bits 101
byte ciphertext[1];

wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesSetKey(&aes, key, 16, iv, AES_ENCRYPTION);
int ret = wc_AesCfb1Encrypt(&aes, ciphertext, plaintext, 3);
if (ret != 0) {
    // encryption failed
}
wc_AesFree(&aes);
```

```
int wc_AesCfb8Encrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function encrypts data using AES CFB-8 mode (8-bit feedback). It processes data one byte at a time, making it suitable for byte-oriented stream encryption.

Parameters:

- **aes** pointer to the AES structure containing the key
- **out** pointer to the output buffer to store encrypted data
- **in** pointer to the input buffer containing data to encrypt
- **sz** size of input in bytes

See:

- wc_AesCfb8Decrypt
- wc_AesCfb1Encrypt

Return:

- 0 On success.
- BAD_FUNC_ARG If aes, out, or in is NULL.
- Other negative values on error.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte iv[16] = { }; // initialization vector
byte plaintext[10] = { }; // data to encrypt
byte ciphertext[10];

wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesSetKey(&aes, key, 16, iv, AES_ENCRYPTION);
int ret = wc_AesCfb8Encrypt(&aes, ciphertext, plaintext, 10);
if (ret != 0) {
    // encryption failed
}
wc_AesFree(&aes);
```

```
int wc_AesCfb1Decrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function decrypts data using AES CFB-1 mode (1-bit feedback). It processes data one bit at a time, making it suitable for bit-oriented applications.

Parameters:

- **aes** pointer to the AES structure containing the key

- **out** pointer to the output buffer to store decrypted data
- **in** pointer to the input buffer containing data to decrypt
- **sz** size of input in bits

See:

- wc_AesCfb1Encrypt
- wc_AesCfb8Decrypt

Return:

- 0 On success.
- BAD_FUNC_ARG If aes, out, or in is NULL.
- Other negative values on error.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte iv[16] = { }; // initialization vector
byte ciphertext[1] = { }; // encrypted bits
byte plaintext[1];

wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesSetKey(&aes, key, 16, iv, AES_ENCRYPTION);
int ret = wc_AesCfb1Decrypt(&aes, plaintext, ciphertext, 3);
if (ret != 0) {
    // decryption failed
}
wc_AesFree(&aes);
```

```
int wc_AesCfb8Decrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function decrypts data using AES CFB-8 mode (8-bit feedback). It processes data one byte at a time, making it suitable for byte-oriented stream decryption.

Parameters:

- **aes** pointer to the AES structure containing the key
- **out** pointer to the output buffer to store decrypted data
- **in** pointer to the input buffer containing data to decrypt
- **sz** size of input in bytes

See:

- wc_AesCfb8Encrypt
- wc_AesCfb1Decrypt

Return:

- 0 On success.
- BAD_FUNC_ARG If aes, out, or in is NULL.
- Other negative values on error.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte iv[16] = { }; // initialization vector
byte ciphertext[10] = { }; // encrypted data
byte plaintext[10];

wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesSetKey(&aes, key, 16, iv, AES_ENCRYPTION);
int ret = wc_AesCfb8Decrypt(&aes, plaintext, ciphertext, 10);
if (ret != 0) {
    // decryption failed
}
wc_AesFree(&aes);
```

```
int wc_AesOfbEncrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function encrypts data using AES OFB mode (Output Feedback). OFB mode turns a block cipher into a stream cipher by encrypting the IV and XORing with plaintext.

Parameters:

- **aes** pointer to the AES structure containing the key
- **out** pointer to the output buffer to store encrypted data
- **in** pointer to the input buffer containing data to encrypt
- **sz** size of input in bytes

See:

- wc_AesOfbDecrypt
- wc_AesSetKey

Return:

- 0 On success.

- BAD_FUNC_ARG If aes, out, or in is NULL.
- Other negative values on error.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte iv[16] = { }; // initialization vector
byte plaintext[100] = { }; // data to encrypt
byte ciphertext[100];

wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesSetKey(&aes, key, 16, iv, AES_ENCRYPTION);
int ret = wc_AesOfbEncrypt(&aes, ciphertext, plaintext, 100);
if (ret != 0) {
    // encryption failed
}
wc_AesFree(&aes);
```

```
int wc_AesOfbDecrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function decrypts data using AES OFB mode (Output Feedback). In OFB mode, encryption and decryption are the same operation.

Parameters:

- **aes** pointer to the AES structure containing the key
- **out** pointer to the output buffer to store decrypted data
- **in** pointer to the input buffer containing data to decrypt
- **sz** size of input in bytes

See:

- wc_AesOfbEncrypt
- wc_AesSetKey

Return:

- 0 On success.
- BAD_FUNC_ARG If aes, out, or in is NULL.
- Other negative values on error.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte iv[16] = { }; // initialization vector
byte ciphertext[100] = { }; // encrypted data
byte plaintext[100];

wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesSetKey(&aes, key, 16, iv, AES_ENCRYPTION);
int ret = wc_AesOfbDecrypt(&aes, plaintext, ciphertext, 100);
if (ret != 0) {
    // decryption failed
}
wc_AesFree(&aes);
```

```
int wc_AesEcbEncrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function encrypts data using AES ECB mode (Electronic Codebook). Warning: ECB mode is not recommended for most use cases as it does not provide semantic security. Each block is encrypted independently.

Parameters:

- **aes** pointer to the AES structure containing the key
- **out** pointer to the output buffer to store encrypted data
- **in** pointer to the input buffer containing data to encrypt
- **sz** size of input in bytes (must be multiple of AES_BLOCK_SIZE)

See:

- wc_AesEcbDecrypt
- wc_AesSetKey

Return:

- 0 On success.
- BAD_FUNC_ARG If aes, out, or in is NULL.
- Other negative values on error.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte plaintext[32] = { }; // data to encrypt
byte ciphertext[32];
```

```
wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesSetKey(&aes, key, 16, NULL, AES_ENCRYPTION);
int ret = wc_AesEcbEncrypt(&aes, ciphertext, plaintext, 32);
if (ret != 0) {
    // encryption failed
}
wc_AesFree(&aes);
```

```
int wc_AesEcbDecrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function decrypts data using AES ECB mode (Electronic Codebook). Warning: ECB mode is not recommended for most use cases as it does not provide semantic security. Each block is decrypted independently.

Parameters:

- **aes** pointer to the AES structure containing the key
- **out** pointer to the output buffer to store decrypted data
- **in** pointer to the input buffer containing data to decrypt
- **sz** size of input in bytes (must be multiple of AES_BLOCK_SIZE)

See:

- wc_AesEcbEncrypt
- wc_AesSetKey

Return:

- 0 On success.
- BAD_FUNC_ARG If aes, out, or in is NULL.
- Other negative values on error.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte ciphertext[32] = { }; // encrypted data
byte plaintext[32];

wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesSetKey(&aes, key, 16, NULL, AES_DECRYPTION);
int ret = wc_AesEcbDecrypt(&aes, plaintext, ciphertext, 32);
if (ret != 0) {
    // decryption failed
```

```
}
wc_AesFree(&aes);
```

```
int wc_AesCtrSetKey(
    Aes * aes,
    const byte * key,
    word32 len,
    const byte * iv,
    int dir
)
```

This function sets the key and IV for AES CTR mode. It initializes the AES structure for counter mode encryption or decryption.

Parameters:

- **aes** pointer to the AES structure to initialize
- **key** pointer to the key buffer (16, 24, or 32 bytes)
- **len** length of the key in bytes
- **iv** pointer to the initialization vector (16 bytes)
- **dir** cipher direction (always use AES_ENCRYPTION for CTR mode)

See:

- wc_AesCtrEncrypt
- wc_AesSetKey

Return:

- 0 On success.
- BAD_FUNC_ARG If aes, key, or iv is NULL, or if key length is invalid.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte iv[16] = { }; // initialization vector

wc_AesInit(&aes, NULL, INVALID_DEVID);
int ret = wc_AesCtrSetKey(&aes, key, 16, iv, AES_ENCRYPTION);
if (ret != 0) {
    // failed to set key
}
wc_AesFree(&aes);
```

```
int wc_AesGcmSetKey_ex(
    Aes * aes,
    const byte * key,
```

```
    word32 len,
    word32 kup
)
```

This function sets the key for AES GCM with an extended key update parameter. It allows for key updates in certain hardware implementations.

Parameters:

- **aes** pointer to the AES structure to initialize
- **key** pointer to the key buffer (16, 24, or 32 bytes)
- **len** length of the key in bytes
- **kup** key update parameter for hardware implementations

See:

- wc_AesGcmSetKey
- wc_AesGcmInit

Return:

- 0 On success.
- BAD_FUNC_ARG If aes or key is NULL, or if key length is invalid.

Note: This function is currently only available when building with Xilinx hardware acceleration. It requires one of the following build options: WOLFSSL_XILINX_CRYPT (for Xilinx SecureIP integration) or WOLFSSL_AFALG_XILINX_AES (for Xilinx AF_ALG support). This API may be exposed for additional build configurations in the future.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key

wc_AesInit(&aes, NULL, INVALID_DEVID);
int ret = wc_AesGcmSetKey_ex(&aes, key, 16, 0);
if (ret != 0) {
    // failed to set key
}
wc_AesFree(&aes);
```

```
int wc_AesGcmInit(
    Aes * aes,
    const byte * key,
    word32 len,
    const byte * iv,
    word32 ivSz
)
```

This function initializes an AES GCM cipher with key and IV. It can be called with NULL key to only set the IV, or with NULL IV to only set the key.

Parameters:

- **aes** pointer to the AES structure to initialize
- **key** pointer to the key buffer, or NULL to skip key setting
- **len** length of the key in bytes
- **iv** pointer to the IV/nonce buffer, or NULL to skip IV setting
- **ivSz** length of the IV/nonce in bytes

See:

- wc_AesGcmSetKey
- wc_AesGcmEncrypt

Return:

- 0 On success.
- BAD_FUNC_ARG If aes is NULL, or if parameters are invalid.
- MEMORY_E If dynamic memory allocation fails.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte iv[12] = { }; // 96-bit nonce

wc_AesInit(&aes, NULL, INVALID_DEVID);
int ret = wc_AesGcmInit(&aes, key, 16, iv, 12);
if (ret != 0) {
    // failed to initialize
}
wc_AesFree(&aes);
```

```
int wc_AesGcmEncryptInit(
    Aes * aes,
    const byte * key,
    word32 len,
    const byte * iv,
    word32 ivSz
)
```

This function initializes an AES GCM cipher for encryption. It is a convenience wrapper around wc_AesGcmInit for encryption operations.

Parameters:

- **aes** pointer to the AES structure to initialize
- **key** pointer to the key buffer, or NULL to skip key setting
- **len** length of the key in bytes
- **iv** pointer to the IV/nonce buffer, or NULL to skip IV setting

- **ivSz** length of the IV/nonce in bytes

See:

- wc_AesGcmInit
- wc_AesGcmEncryptUpdate

Return:

- 0 On success.
- BAD_FUNC_ARG If aes is NULL, or if parameters are invalid.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte iv[12] = { }; // 96-bit nonce

wc_AesInit(&aes, NULL, INVALID_DEVID);
int ret = wc_AesGcmEncryptInit(&aes, key, 16, iv, 12);
if (ret != 0) {
    // failed to initialize
}
wc_AesFree(&aes);
```

```
int wc_AesGcmEncryptInit_ex(
    Aes * aes,
    const byte * key,
    word32 len,
    byte * ivOut,
    word32 ivOutSz
)
```

This function initializes an AES GCM cipher for encryption and outputs the IV. This is useful when part of the IV is generated internally. Must call wc_AesGcmSetIV() before this function to set the fixed part of the IV.

Parameters:

- **aes** pointer to the AES structure to initialize
- **key** pointer to the key buffer, or NULL to skip key setting
- **len** length of the key in bytes
- **ivOut** pointer to buffer to receive the complete IV
- **ivOutSz** length of the IV output buffer in bytes

See:

- wc_AesGcmSetIV
- wc_AesGcmEncryptUpdate

Return:

- 0 On success.
- BAD_FUNC_ARG If aes, ivOut is NULL, or if ivOutSz doesn't match the cached nonce size.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte ivFixed[4] = { }; // fixed part of IV
byte ivOut[12];
WC_RNG rng;

wc_InitRng(&rng);
wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesGcmSetIV(&aes, 12, ivFixed, 4, &rng);
int ret = wc_AesGcmEncryptInit_ex(&aes, key, 16, ivOut, 12);
if (ret != 0) {
    // failed to initialize
}
wc_AesFree(&aes);
wc_FreeRng(&rng);
```

```
int wc_AesGcmEncryptUpdate(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    const byte * authIn,
    word32 authInSz
)
```

This function performs an update step of AES GCM encryption. It processes plaintext and/or additional authentication data (AAD) in a streaming fashion.

Parameters:

- **aes** pointer to the AES structure
- **out** pointer to buffer to store ciphertext (can be NULL if sz=0)
- **in** pointer to plaintext to encrypt (can be NULL if sz=0)
- **sz** length of plaintext in bytes
- **authIn** pointer to additional authentication data (can be NULL)
- **authInSz** length of AAD in bytes

See:

- wc_AesGcmInit
- wc_AesGcmEncryptInit
- wc_AesGcmEncryptFinal

Return:

- 0 On success.
- BAD_FUNC_ARG If aes is NULL, or a length is non-zero but buffer is NULL.

All the AAD must be passed to update before the plaintext. The last part of AAD can be passed with the first part of plaintext.

Must set key and IV before calling this function. Must call wc_AesGcmInit() before calling this function.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte iv[12] = { }; // nonce
byte plaintext[100] = { }; // data
byte ciphertext[100];
byte aad[20] = { }; // additional data

wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesGcmInit(&aes, key, 16, iv, 12);
int ret = wc_AesGcmEncryptUpdate(&aes, ciphertext, plaintext, 100,
                                 aad, 20);
if (ret != 0) {
    // encryption failed
}
wc_AesFree(&aes);
```

```
int wc_AesGcmEncryptFinal(
    Aes * aes,
    byte * authTag,
    word32 authTagSz
)
```

This function finalizes AES GCM encryption and generates the authentication tag. This must be called after all data has been processed with wc_AesGcmEncryptUpdate.

Parameters:

- **aes** pointer to the AES structure
- **authTag** pointer to buffer to store the authentication tag
- **authTagSz** length of the authentication tag in bytes (typically 12 or 16)

See:

- wc_AesGcmEncryptUpdate
- wc_AesGcmDecryptFinal

Return:

- 0 On success.
- BAD_FUNC_ARG If aes or authTag is NULL, or if authTagSz is invalid.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte iv[12] = { }; // nonce
byte plaintext[100] = { }; // data
byte ciphertext[100];
byte authTag[16];

wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesGcmEncryptInit(&aes, key, 16, iv, 12);
wc_AesGcmEncryptUpdate(&aes, ciphertext, plaintext, 100, NULL, 0);
int ret = wc_AesGcmEncryptFinal(&aes, authTag, 16);
if (ret != 0) {
    // failed to generate tag
}
wc_AesFree(&aes);
```

```
int wc_AesGcmDecryptInit(
    Aes * aes,
    const byte * key,
    word32 len,
    const byte * iv,
    word32 ivSz
)
```

This function initializes an AES GCM cipher for decryption. It is a convenience wrapper around wc_AesGcmInit for decryption operations.

Parameters:

- **aes** pointer to the AES structure to initialize
- **key** pointer to the key buffer, or NULL to skip key setting
- **len** length of the key in bytes
- **iv** pointer to the IV/nonce buffer, or NULL to skip IV setting
- **ivSz** length of the IV/nonce in bytes

See:

- wc_AesGcmInit
- wc_AesGcmDecryptUpdate

Return:

- 0 On success.
- BAD_FUNC_ARG If aes is NULL, or if parameters are invalid.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte iv[12] = { }; // 96-bit nonce

wc_AesInit(&aes, NULL, INVALID_DEVID);
int ret = wc_AesGcmDecryptInit(&aes, key, 16, iv, 12);
if (ret != 0) {
    // failed to initialize
}
wc_AesFree(&aes);
```

```
int wc_AesGcmDecryptUpdate(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    const byte * authIn,
    word32 authInSz
)
```

This function performs an update step of AES GCM decryption. It processes ciphertext and/or additional authentication data (AAD) in a streaming fashion.

Parameters:

- **aes** pointer to the AES structure
- **out** pointer to buffer to store plaintext (can be NULL if sz=0)
- **in** pointer to ciphertext to decrypt (can be NULL if sz=0)
- **sz** length of ciphertext in bytes
- **authIn** pointer to additional authentication data (can be NULL)
- **authInSz** length of AAD in bytes

See:

- wc_AesGcmInit
- wc_AesGcmDecryptInit
- wc_AesGcmDecryptFinal

Return:

- 0 On success.
- BAD_FUNC_ARG If aes is NULL, or a length is non-zero but buffer is NULL.

All the AAD must be passed to update before the ciphertext. The last part of AAD can be passed with the first part of ciphertext.

Must set key and IV before calling this function. Must call wc_AesGcmInit() before calling this function.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte iv[12] = { }; // nonce
byte ciphertext[100] = { }; // encrypted data
byte plaintext[100];
byte aad[20] = { }; // additional data

wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesGcmInit(&aes, key, 16, iv, 12);
int ret = wc_AesGcmDecryptUpdate(&aes, plaintext, ciphertext, 100,
                                 aad, 20);
if (ret != 0) {
    // decryption failed
}
wc_AesFree(&aes);
```

```
int wc_AesGcmDecryptFinal(
    Aes * aes,
    const byte * authTag,
    word32 authTagSz
)
```

This function finalizes AES GCM decryption and verifies the authentication tag. This must be called after all data has been processed with wc_AesGcmDecryptUpdate.

Parameters:

- **aes** pointer to the AES structure
- **authTag** pointer to the authentication tag to verify
- **authTagSz** length of the authentication tag in bytes

See:

- wc_AesGcmDecryptUpdate
- wc_AesGcmEncryptFinal

Return:

- 0 On success.
- AES_GCM_AUTH_E If authentication tag verification fails.
- BAD_FUNC_ARG If aes or authTag is NULL, or if authTagSz is invalid.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte iv[12] = { }; // nonce
byte ciphertext[100] = { }; // encrypted data
byte plaintext[100];
```

```
byte authTag[16] = { }; // received tag

wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesGcmDecryptInit(&aes, key, 16, iv, 12);
wc_AesGcmDecryptUpdate(&aes, plaintext, ciphertext, 100, NULL, 0);
int ret = wc_AesGcmDecryptFinal(&aes, authTag, 16);
if (ret != 0) {
    // authentication failed
}
wc_AesFree(&aes);
```

```
int wc_AesGcmSetExtIV(
    Aes * aes,
    const byte * iv,
    word32 ivSz
)
```

This function sets an external IV for AES GCM. This allows using an IV that was generated externally or received from another source.

Parameters:

- **aes** pointer to the AES structure
- **iv** pointer to the IV/nonce buffer
- **ivSz** length of the IV/nonce in bytes

See:

- wc_AesGcmSetIV
- wc_AesGcmInit

Return:

- 0 On success.
- BAD_FUNC_ARG If aes or iv is NULL, or if ivSz is invalid.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte iv[12] = { }; // external nonce

wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesGcmSetKey(&aes, key, 16);
int ret = wc_AesGcmSetExtIV(&aes, iv, 12);
if (ret != 0) {
    // failed to set IV
}
wc_AesFree(&aes);
```

```
int wc_AesGcmSetIV(
    Aes * aes,
    word32 ivSz,
    const byte * ivFixed,
    word32 ivFixedSz,
    WC_RNG * rng
)
```

This function sets the IV for AES GCM with optional random generation. It can generate part of the IV using an RNG, which is useful for ensuring IV uniqueness.

Parameters:

- **aes** pointer to the AES structure
- **ivSz** total length of the IV/nonce in bytes
- **ivFixed** pointer to the fixed part of the IV (can be NULL)
- **ivFixedSz** length of the fixed part in bytes
- **rng** pointer to initialized RNG for generating random part (can be NULL if ivFixedSz equals ivSz)

See:

- wc_AesGcmSetExtIV
- wc_AesGcmEncryptInit_ex

Return:

- 0 On success.
- BAD_FUNC_ARG If aes is NULL, or if parameters are invalid.
- Other negative values on RNG or other errors.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte ivFixed[4] = { }; // fixed part
WC_RNG rng;

wc_InitRng(&rng);
wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesGcmSetKey(&aes, key, 16);
int ret = wc_AesGcmSetIV(&aes, 12, ivFixed, 4, &rng);
if (ret != 0) {
    // failed to set IV
}
wc_AesFree(&aes);
wc_FreeRng(&rng);
```

```
int wc_AesGcmEncrypt_ex(
    Aes * aes,
    byte * out,
```

```
    const byte * in,
    word32 sz,
    byte * ivOut,
    word32 ivOutSz,
    byte * authTag,
    word32 authTagSz,
    const byte * authIn,
    word32 authInSz
)
```

This function performs AES GCM encryption with extended parameters, including IV output. This is a one-shot encryption function that outputs the generated IV.

Parameters:

- **aes** pointer to the AES structure
- **out** pointer to buffer to store ciphertext
- **in** pointer to plaintext to encrypt
- **sz** length of plaintext in bytes
- **ivOut** pointer to buffer to receive the IV
- **ivOutSz** length of the IV output buffer in bytes
- **authTag** pointer to buffer to store authentication tag
- **authTagSz** length of authentication tag in bytes
- **authIn** pointer to additional authentication data
- **authInSz** length of AAD in bytes

See:

- wc_AesGcmEncrypt
- wc_AesGcmSetIV

Return:

- 0 On success.
- BAD_FUNC_ARG If parameters are invalid.
- WC_PENDING_E If submitted to an asynchronous device and still in progress.
- Other negative values on error.

The IV is taken from an internal counter that is advanced whenever an IV is consumed - on success, and on submission to an asynchronous device - so no two encryptions under one key use the same IV. A call that fails before the cipher consumes the IV leaves the counter where it was. ivOut must not overlap out, in, authTag or authIn: the buffer is the working copy of the IV that was consumed, and overwriting it corrupts the counter for the next call. ivOut may be written even when the function returns an error.

When the Aes carries an asynchronous device, a return of WC_PENDING_E means the operation was submitted, not that it finished. ivOut - like out, in, authTag and authIn - must stay allocated and unmodified until the operation completes, because the backend reads the IV from it at completion time.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
```

```
byte ivFixed[4] = { }; // fixed part
byte ivOut[12];
byte plaintext[100] = { }; // data
byte ciphertext[100];
byte authTag[16];
WC_RNG rng;

wc_InitRng(&rng);
wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesGcmSetKey(&aes, key, 16);
wc_AesGcmSetIV(&aes, 12, ivFixed, 4, &rng);
int ret = wc_AesGcmEncrypt_ex(&aes, ciphertext, plaintext, 100,
                              ivOut, 12, authTag, 16, NULL, 0);
if (ret != 0) {
    // encryption failed
}
wc_AesFree(&aes);
wc_FreeRng(&rng);
```

```
int wc_Gmac(
    const byte * key,
    word32 keySz,
    byte * iv,
    word32 ivSz,
    const byte * authIn,
    word32 authInSz,
    byte * authTag,
    word32 authTagSz,
    WC_RNG * rng
)
```

This function performs GMAC (Galois Message Authentication Code) generation. GMAC is essentially AES-GCM with no plaintext, used for authentication only.

Parameters:

- **key** pointer to the key buffer
- **keySz** length of the key in bytes (16, 24, or 32)
- **iv** pointer to the IV/nonce buffer
- **ivSz** length of the IV/nonce in bytes
- **authIn** pointer to data to authenticate
- **authInSz** length of data to authenticate in bytes
- **authTag** pointer to buffer to store authentication tag
- **authTagSz** length of authentication tag in bytes
- **rng** pointer to initialized RNG (can be NULL if IV is complete)

See:

- wc_GmacVerify
- wc_AesGcmEncrypt

Return:

- 0 On success.
- BAD_FUNC_ARG If parameters are invalid.
- Other negative values on error.

Example

```
byte key[16] = { }; // 128-bit key
byte iv[12] = { }; // nonce
byte data[100] = { }; // data to authenticate
byte authTag[16];

int ret = wc_Gmac(key, 16, iv, 12, data, 100, authTag, 16, NULL);
if (ret != 0) {
    // GMAC generation failed
}
```

```
int wc_GmacVerify(
    const byte * key,
    word32 keySz,
    const byte * iv,
    word32 ivSz,
    const byte * authIn,
    word32 authInSz,
    const byte * authTag,
    word32 authTagSz
)
```

This function verifies a GMAC (Galois Message Authentication Code). It computes the GMAC and compares it with the provided tag.

Parameters:

- **key** pointer to the key buffer
- **keySz** length of the key in bytes (16, 24, or 32)
- **iv** pointer to the IV/nonce buffer
- **ivSz** length of the IV/nonce in bytes
- **authIn** pointer to data to authenticate
- **authInSz** length of data to authenticate in bytes
- **authTag** pointer to the authentication tag to verify
- **authTagSz** length of authentication tag in bytes

See:

- wc_Gmac
- wc_AesGcmDecrypt

Return:

- 0 On successful verification.

- AES_GCM_AUTH_E If authentication tag verification fails.
- BAD_FUNC_ARG If parameters are invalid.
- Other negative values on error.

Example

```
byte key[16] = { }; // 128-bit key
byte iv[12] = { }; // nonce
byte data[100] = { }; // data to authenticate
byte authTag[16] = { }; // received tag

int ret = wc_GmacVerify(key, 16, iv, 12, data, 100, authTag, 16);
if (ret != 0) {
    // GMAC verification failed
}
```

```
int wc_AesCcmSetNonce(
    Aes * aes,
    const byte * nonce,
    word32 nonceSz
)
```

This function sets the nonce for AES CCM mode. The nonce must be set before encryption or decryption operations.

Parameters:

- **aes** pointer to the AES structure
- **nonce** pointer to the nonce buffer
- **nonceSz** length of the nonce in bytes (7-13 bytes for CCM)

See:

- wc_AesCcmEncrypt
- wc_AesCcmSetKey

Return:

- 0 On success.
- BAD_FUNC_ARG If aes or nonce is NULL, or if nonceSz is invalid.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte nonce[12] = { }; // nonce

wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesCcmSetKey(&aes, key, 16);
int ret = wc_AesCcmSetNonce(&aes, nonce, 12);
```

```
if (ret != 0) {
    // failed to set nonce
}
wc_AesFree(&aes);
```

```
int wc_AesCcmEncrypt_ex(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    byte * ivOut,
    word32 ivOutSz,
    byte * authTag,
    word32 authTagSz,
    const byte * authIn,
    word32 authInSz
)
```

This function performs AES CCM encryption with extended parameters, including nonce output. This is useful when part of the nonce is generated internally.

Parameters:

- **aes** pointer to the AES structure
- **out** pointer to buffer to store ciphertext
- **in** pointer to plaintext to encrypt
- **sz** length of plaintext in bytes
- **ivOut** pointer to buffer to receive the nonce
- **ivOutSz** length of the nonce output buffer in bytes
- **authTag** pointer to buffer to store authentication tag
- **authTagSz** length of authentication tag in bytes
- **authIn** pointer to additional authentication data
- **authInSz** length of AAD in bytes

See:

- wc_AesCcmEncrypt
- wc_AesCcmSetNonce

Return:

- 0 On success.
- BAD_FUNC_ARG If parameters are invalid.
- Other negative values on error.

The nonce is taken from an internal counter that is advanced whenever a nonce is consumed - on success, and on submission to an asynchronous device - so no two encryptions under one key use the same nonce. A call that fails before the cipher consumes the nonce leaves the counter where it was. ivOut must not overlap out, in, authTag or authIn: the buffer is the working copy of the nonce that was consumed, and overwriting it corrupts the counter for the next call. ivOut may be written even when the function returns an error - earlier releases left it untouched on failure.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte nonce[12];
byte plaintext[100] = { }; // data
byte ciphertext[100];
byte authTag[16];

wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesCcmSetKey(&aes, key, 16);
int ret = wc_AesCcmEncrypt_ex(&aes, ciphertext, plaintext, 100,
                              nonce, 12, authTag, 16, NULL, 0);
if (ret != 0) {
    // encryption failed
}
wc_AesFree(&aes);
```

```
int wc_AesKeyWrap(
    const byte * key,
    word32 keySz,
    const byte * in,
    word32 inSz,
    byte * out,
    word32 outSz,
    const byte * iv
)
```

This function wraps a key using AES Key Wrap algorithm (RFC 3394). This is commonly used to securely transport cryptographic keys.

Parameters:

- **key** pointer to the key-encryption key
- **keySz** length of the key-encryption key in bytes
- **in** pointer to the key to wrap
- **inSz** length of the key to wrap in bytes
- **out** pointer to buffer to store wrapped key
- **outSz** size of output buffer in bytes
- **iv** pointer to IV (typically NULL to use default)

See:

- wc_AesKeyUnWrap
- wc_AesKeyWrap_ex

Return:

- Length of wrapped key in bytes on success.
- BAD_FUNC_ARG If parameters are invalid.

- Other negative values on error.

Example

```
byte kek[16] = { }; // key-encryption key
byte keyToWrap[16] = { }; // key to wrap
byte wrappedKey[24];

int wrappedLen = wc_AesKeyWrap(kek, 16, keyToWrap, 16, wrappedKey,
                               24, NULL);
if (wrappedLen <= 0) {
    // key wrap failed
}
```

```
int wc_AesKeyWrap_ex(
    Aes * aes,
    const byte * in,
    word32 inSz,
    byte * out,
    word32 outSz,
    const byte * iv
)
```

This function wraps a key using AES Key Wrap algorithm with an initialized AES structure. This allows reusing the same AES structure for multiple wrap operations.

Parameters:

- **aes** pointer to initialized AES structure
- **in** pointer to the key to wrap
- **inSz** length of the key to wrap in bytes
- **out** pointer to buffer to store wrapped key
- **outSz** size of output buffer in bytes
- **iv** pointer to IV (typically NULL to use default)

See:

- wc_AesKeyWrap
- wc_AesKeyUnWrap_ex

Return:

- Length of wrapped key in bytes on success.
- BAD_FUNC_ARG If parameters are invalid.
- Other negative values on error.

Example

```
Aes aes;
byte kek[16] = { }; // key-encryption key
byte keyToWrap[16] = { }; // key to wrap
byte wrappedKey[24];

wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesSetKey(&aes, kek, 16, NULL, AES_ENCRYPTION);
int wrappedLen = wc_AesKeyWrap_ex(&aes, keyToWrap, 16, wrappedKey,
                                  24, NULL);
if (wrappedLen <= 0) {
    // key wrap failed
}
wc_AesFree(&aes);
```

```
int wc_AesKeyUnWrap(
    const byte * key,
    word32 keySz,
    const byte * in,
    word32 inSz,
    byte * out,
    word32 outSz,
    const byte * iv
)
```

This function unwraps a key using AES Key Unwrap algorithm (RFC 3394). This is used to securely receive cryptographic keys that were wrapped.

Parameters:

- **key** pointer to the key-encryption key
- **keySz** length of the key-encryption key in bytes
- **in** pointer to the wrapped key
- **inSz** length of the wrapped key in bytes
- **out** pointer to buffer to store unwrapped key
- **outSz** size of output buffer in bytes
- **iv** pointer to IV (typically NULL to use default)

See:

- wc_AesKeyWrap
- wc_AesKeyUnWrap_ex

Return:

- Length of unwrapped key in bytes on success.
- BAD_FUNC_ARG If parameters are invalid.
- Other negative values on error.

Example

```
byte kek[16] = { }; // key-encryption key
byte wrappedKey[24] = { }; // wrapped key
byte unwrappedKey[16];

int unwrappedLen = wc_AesKeyUnWrap(kek, 16, wrappedKey, 24,
                                   unwrappedKey, 16, NULL);
if (unwrappedLen <= 0) {
    // key unwrap failed
}
```

```
int wc_AesKeyUnWrap_ex(
    Aes * aes,
    const byte * in,
    word32 inSz,
    byte * out,
    word32 outSz,
    const byte * iv
)
```

This function unwraps a key using AES Key Unwrap algorithm with an initialized AES structure. This allows reusing the same AES structure for multiple unwrap operations.

Parameters:

- **aes** pointer to initialized AES structure
- **in** pointer to the wrapped key
- **inSz** length of the wrapped key in bytes
- **out** pointer to buffer to store unwrapped key
- **outSz** size of output buffer in bytes
- **iv** pointer to IV (typically NULL to use default)

See:

- wc_AesKeyUnWrap
- wc_AesKeyWrap_ex

Return:

- Length of unwrapped key in bytes on success.
- BAD_FUNC_ARG If parameters are invalid.
- Other negative values on error.

Example

```
Aes aes;
byte kek[16] = { }; // key-encryption key
byte wrappedKey[24] = { }; // wrapped key
byte unwrappedKey[16];

wc_AesInit(&aes, NULL, INVALID_DEVID);
```

```
wc_AesSetKey(&aes, kek, 16, NULL, AES_ENCRYPTION);
int unwrappedLen = wc_AesKeyUnWrap_ex(&aes, wrappedKey, 24,
                                      unwrappedKey, 16, NULL);
if (unwrappedLen <= 0) {
    // key unwrap failed
}
wc_AesFree(&aes);
```

```
int wc_AesXtsEncryptConsecutiveSectors(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    word64 sector,
    word32 sectorSz
)
```

This function encrypts multiple consecutive sectors using AES XTS mode. It processes multiple sectors in sequence, automatically incrementing the sector number for each sector.

Parameters:

- **aes** pointer to the XtsAes structure
- **out** pointer to buffer to store encrypted data
- **in** pointer to plaintext data to encrypt
- **sz** total length of data in bytes
- **sector** starting sector number for the tweak
- **sectorSz** size of each sector in bytes

See:

- wc_AesXtsDecryptConsecutiveSectors
- wc_AesXtsEncryptSector

Return:

- 0 On success.
- BAD_FUNC_ARG If aes, out, or in is NULL, or if sectorSz is 0, or if sz is less than AES_BLOCK_SIZE.
- Other negative values on error.

Example

```
XtsAes aes;
byte key[32] = { }; // 256-bit key
byte plaintext[1024] = { }; // data
byte ciphertext[1024];

wc_AesXtsSetKey(&aes, key, 32, AES_ENCRYPTION, NULL, INVALID_DEVID);
int ret = wc_AesXtsEncryptConsecutiveSectors(&aes, ciphertext,
                                             plaintext, 1024, 0, 512);
```

```
if (ret != 0) {
    // encryption failed
}
wc_AesXtsFree(&aes);
```

```
int wc_AesXtsDecryptConsecutiveSectors(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    word64 sector,
    word32 sectorSz
)
```

This function decrypts multiple consecutive sectors using AES XTS mode. It processes multiple sectors in sequence, automatically incrementing the sector number for each sector.

Parameters:

- **aes** pointer to the XtsAes structure
- **out** pointer to buffer to store decrypted data
- **in** pointer to ciphertext data to decrypt
- **sz** total length of data in bytes
- **sector** starting sector number for the tweak
- **sectorSz** size of each sector in bytes

See:

- wc_AesXtsEncryptConsecutiveSectors
- wc_AesXtsDecryptSector

Return:

- 0 On success.
- BAD_FUNC_ARG If aes, out, or in is NULL, or if sectorSz is 0, or if sz is less than AES_BLOCK_SIZE.
- Other negative values on error.

Example

```
XtsAes aes;
byte key[32] = { }; // 256-bit key
byte ciphertext[1024] = { }; // encrypted data
byte plaintext[1024];

wc_AesXtsSetKey(&aes, key, 32, AES_DECRYPTION, NULL, INVALID_DEVID);
int ret = wc_AesXtsDecryptConsecutiveSectors(&aes, plaintext,
                                             ciphertext, 1024, 0, 512);
if (ret != 0) {
    // decryption failed
```

```
}
wc_AesXtsFree(&aes);
```

```
int wc_AesXtsEncryptInit(
    XtsAes * aes,
    const byte * i,
    word32 iSz,
    struct XtsAesStreamData * stream
)
```

This function initializes streaming AES XTS encryption. It sets up the context for processing data in multiple update calls.

Parameters:

- **aes** pointer to the XtsAes structure
- **i** pointer to the tweak/IV buffer
- **iSz** length of the tweak/IV in bytes
- **stream** pointer to XtsAesStreamData structure for streaming state

See:

- wc_AesXtsEncryptUpdate
- wc_AesXtsEncryptFinal

Return:

- 0 On success.
- BAD_FUNC_ARG If parameters are invalid.

Example

```
XtsAes aes;
struct XtsAesStreamData stream;
byte key[32] = { }; // 256-bit key
byte tweak[16] = { }; // tweak value

wc_AesXtsSetKey(&aes, key, 32, AES_ENCRYPTION, NULL, INVALID_DEVID);
int ret = wc_AesXtsEncryptInit(&aes, tweak, 16, &stream);
if (ret != 0) {
    // initialization failed
}
wc_AesXtsFree(&aes);
```

```
int wc_AesXtsDecryptInit(
    XtsAes * aes,
    const byte * i,
    word32 iSz,
```

```
    struct XtsAesStreamData * stream
)
```

This function initializes streaming AES XTS decryption. It sets up the context for processing data in multiple update calls.

Parameters:

- **aes** pointer to the XtsAes structure
- **i** pointer to the tweak/IV buffer
- **iSz** length of the tweak/IV in bytes
- **stream** pointer to XtsAesStreamData structure for streaming state

See:

- wc_AesXtsDecryptUpdate
- wc_AesXtsDecryptFinal

Return:

- 0 On success.
- BAD_FUNC_ARG If parameters are invalid.

Example

```
XtsAes aes;
struct XtsAesStreamData stream;
byte key[32] = { }; // 256-bit key
byte tweak[16] = { }; // tweak value

wc_AesXtsSetKey(&aes, key, 32, AES_DECRYPTION, NULL, INVALID_DEVID);
int ret = wc_AesXtsDecryptInit(&aes, tweak, 16, &stream);
if (ret != 0) {
    // initialization failed
}
wc_AesXtsFree(&aes);
```

```
int wc_AesXtsEncryptUpdate(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    struct XtsAesStreamData * stream
)
```

This function performs an update step of streaming AES XTS encryption. It processes a chunk of data and can be called multiple times.

Parameters:

- **aes** pointer to the XtsAes structure
- **out** pointer to buffer to store encrypted data
- **in** pointer to plaintext data to encrypt
- **sz** length of data in bytes
- **stream** pointer to XtsAesStreamData structure for streaming state

See:

- wc_AesXtsEncryptInit
- wc_AesXtsEncryptFinal

Return:

- 0 On success.
- BAD_FUNC_ARG If parameters are invalid.

Example

```
XtsAes aes;
struct XtsAesStreamData stream;
byte key[32] = { }; // 256-bit key
byte tweak[16] = { }; // tweak value
byte plaintext[100] = { }; // data
byte ciphertext[100];

wc_AesXtsSetKey(&aes, key, 32, AES_ENCRYPTION, NULL, INVALID_DEVID);
wc_AesXtsEncryptInit(&aes, tweak, 16, &stream);
int ret = wc_AesXtsEncryptUpdate(&aes, ciphertext, plaintext, 100,
                                 &stream);
if (ret != 0) {
    // encryption failed
}
wc_AesXtsFree(&aes);
```

```
int wc_AesXtsDecryptUpdate(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    struct XtsAesStreamData * stream
)
```

This function performs an update step of streaming AES XTS decryption. It processes a chunk of data and can be called multiple times.

Parameters:

- **aes** pointer to the XtsAes structure
- **out** pointer to buffer to store decrypted data
- **in** pointer to ciphertext data to decrypt

- **sz** length of data in bytes
- **stream** pointer to XtsAesStreamData structure for streaming state

See:

- wc_AesXtsDecryptInit
- wc_AesXtsDecryptFinal

Return:

- 0 On success.
- BAD_FUNC_ARG If parameters are invalid.

Example

```
XtsAes aes;
struct XtsAesStreamData stream;
byte key[32] = { }; // 256-bit key
byte tweak[16] = { }; // tweak value
byte ciphertext[100] = { }; // encrypted data
byte plaintext[100];

wc_AesXtsSetKey(&aes, key, 32, AES_DECRYPTION, NULL, INVALID_DEVID);
wc_AesXtsDecryptInit(&aes, tweak, 16, &stream);
int ret = wc_AesXtsDecryptUpdate(&aes, plaintext, ciphertext, 100,
                                 &stream);
if (ret != 0) {
    // decryption failed
}
wc_AesXtsFree(&aes);
```

```
int wc_AesXtsEncryptFinal(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    struct XtsAesStreamData * stream
)
```

This function finalizes streaming AES XTS encryption. It processes any remaining data and completes the encryption operation.

Parameters:

- **aes** pointer to the XtsAes structure
- **out** pointer to buffer to store final encrypted data
- **in** pointer to final plaintext data to encrypt
- **sz** length of final data in bytes
- **stream** pointer to XtsAesStreamData structure for streaming state

See:

- wc_AesXtsEncryptUpdate
- wc_AesXtsEncryptInit

Return:

- 0 On success.
- BAD_FUNC_ARG If parameters are invalid.

Example

```
XtsAes aes;
struct XtsAesStreamData stream;
byte key[32] = { }; // 256-bit key
byte tweak[16] = { }; // tweak value
byte plaintext[50] = { }; // final data
byte ciphertext[50];

wc_AesXtsSetKey(&aes, key, 32, AES_ENCRYPTION, NULL, INVALID_DEVID);
wc_AesXtsEncryptInit(&aes, tweak, 16, &stream);
// ... update calls ...
int ret = wc_AesXtsEncryptFinal(&aes, ciphertext, plaintext, 50,
                                &stream);
if (ret != 0) {
    // finalization failed
}
wc_AesXtsFree(&aes);
```

```
int wc_AesXtsDecryptFinal(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    struct XtsAesStreamData * stream
)
```

This function finalizes streaming AES XTS decryption. It processes any remaining data and completes the decryption operation.

Parameters:

- **aes** pointer to the XtsAes structure
- **out** pointer to buffer to store final decrypted data
- **in** pointer to final ciphertext data to decrypt
- **sz** length of final data in bytes
- **stream** pointer to XtsAesStreamData structure for streaming state

See:

- wc_AesXtsDecryptUpdate
- wc_AesXtsDecryptInit

Return:

- 0 On success.
- BAD_FUNC_ARG If parameters are invalid.

Example

```
XtsAes aes;
struct XtsAesStreamData stream;
byte key[32] = { }; // 256-bit key
byte tweak[16] = { }; // tweak value
byte ciphertext[50] = { }; // final encrypted data
byte plaintext[50];

wc_AesXtsSetKey(&aes, key, 32, AES_DECRYPTION, NULL, INVALID_DEVID);
wc_AesXtsDecryptInit(&aes, tweak, 16, &stream);
// ... update calls ...
int ret = wc_AesXtsDecryptFinal(&aes, plaintext, ciphertext, 50,
                                &stream);
if (ret != 0) {
    // finalization failed
}
wc_AesXtsFree(&aes);
```

```
int wc_AesGetKeySize(
    Aes * aes,
    word32 * keySize
)
```

This function retrieves the key size from an initialized AES structure. It returns the size of the key currently set in the AES object.

Parameters:

- **aes** pointer to the AES structure
- **keySize** pointer to word32 to store the key size in bytes

See:

- wc_AesSetKey
- wc_AesInit

Return:

- 0 On success.
- BAD_FUNC_ARG If aes or keySize is NULL.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
word32 keySize;

wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesSetKey(&aes, key, 16, NULL, AES_ENCRYPTION);
int ret = wc_AesGetKeySize(&aes, &keySize);
if (ret == 0) {
    // keySize now contains 16
}
wc_AesFree(&aes);
```

```
int wc_AesInit_Id(
    Aes * aes,
    unsigned char * id,
    int len,
    void * heap,
    int devId
)
```

This function initializes an AES structure with an ID. This is useful for tracking or identifying specific AES instances in applications that manage multiple AES contexts.

Parameters:

- **aes** pointer to the AES structure to initialize
- **id** pointer to the ID buffer
- **len** length of the ID in bytes
- **heap** pointer to heap hint for memory allocation (can be NULL)
- **devId** device ID for hardware acceleration (use INVALID_DEVID for software)

See:

- wc_AesInit
- wc_AesInit_Label

Return:

- 0 On success.
- BAD_FUNC_ARG If aes is NULL, or if id is NULL while len is positive.
- BUFFER_E If len is negative or greater than AES_MAX_ID_LEN.

Note: This API is only available when WOLF_PRIVATE_KEY_ID is defined, which is set for PKCS11 support.

Example

```
Aes aes;
byte id[8] = { }; // unique identifier

int ret = wc_AesInit_Id(&aes, id, 8, NULL, INVALID_DEVID);
if (ret != 0) {
    // initialization failed
}
wc_AesFree(&aes);
```

```
int wc_AesInit_Label(
    Aes * aes,
    const char * label,
    void * heap,
    int devId
)
```

This function initializes an AES structure with a label string. This is useful for tracking or identifying specific AES instances with human-readable names.

Parameters:

- **aes** pointer to the AES structure to initialize
- **label** pointer to the null-terminated label string
- **heap** pointer to heap hint for memory allocation (can be NULL)
- **devId** device ID for hardware acceleration (use INVALID_DEVID for software)

See:

- wc_AesInit
- wc_AesInit_Id

Return:

- 0 On success.
- BAD_FUNC_ARG If aes or label is NULL.

Note: This API is only available when WOLF_PRIVATE_KEY_ID is defined, which is set for PKCS11 support.

Example

```
Aes aes;

int ret = wc_AesInit_Label(&aes, "MyAESContext", NULL, INVALID_DEVID);
if (ret != 0) {
    // initialization failed
}
wc_AesFree(&aes);
```

```
Aes * wc_AesNew(
    void * heap,
    int devId,
    int * result_code
)
```

This function allocates and initializes a new AES structure. It returns a pointer to the allocated structure, which must be freed with wc_AesDelete when no longer needed. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.

Parameters:

- **heap** pointer to heap hint for memory allocation (can be NULL)
- **devId** device ID for hardware acceleration (use INVALID_DEVID for software)
- **result_code** pointer to int to store result code (can be NULL)

See:

- wc_AesDelete
- wc_AesInit

Return:

- Pointer to allocated Aes structure on success.
- NULL on allocation failure.

Note: This API is only available when WC_NO_CONSTRUCTORS is not defined. WC_NO_CONSTRUCTORS is automatically defined when WOLFSSL_NO_MALLOC is defined.

Example

```
int result;
Aes* aes = wc_AesNew(NULL, INVALID_DEVID, &result);
if (aes == NULL || result != 0) {
    // allocation or initialization failed
}
// use aes...
wc_AesDelete(aes, &aes);
```

```
int wc_AesDelete(
    Aes * aes,
    Aes ** aes_p
)
```

This function frees an AES structure that was allocated with wc_AesNew. It also sets the pointer to NULL to prevent use-after-free. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.

Parameters:

- **aes** pointer to the AES structure to free
- **aes_p** pointer to the AES pointer (will be set to NULL)

See:

- wc_AesNew
- wc_AesFree

Return:

- 0 On success.
- BAD_FUNC_ARG If aes or aes_p is NULL.

Note: This API is only available when WC_NO_CONSTRUCTORS is not defined. WC_NO_CONSTRUCTORS is automatically defined when WOLFSSL_NO_MALLOC is defined.

Example

```
Aes* aes = wc_AesNew(NULL, INVALID_DEVID, NULL);
if (aes != NULL) {
    // use aes...
    int ret = wc_AesDelete(aes, &aes);
    // aes is now NULL
}
```

```
int wc_AesSivEncrypt_ex(
    const byte * key,
    word32 keySz,
    const AesSivAssoc * assoc,
    word32 numAssoc,
    const byte * nonce,
    word32 nonceSz,
    const byte * in,
    word32 inSz,
    byte * siv,
    byte * out
)
```

This function performs AES-SIV (Synthetic IV) encryption with extended parameters. AES-SIV provides nonce-misuse resistance and deterministic authenticated encryption.

Parameters:

- **key** pointer to the key buffer (32, 48, or 64 bytes for SIV)
- **keySz** length of the key in bytes
- **assoc** pointer to array of associated data structures
- **numAssoc** number of associated data items
- **nonce** pointer to the nonce buffer (can be NULL)
- **nonceSz** length of the nonce in bytes
- **in** pointer to plaintext to encrypt
- **inSz** length of plaintext in bytes

- **siv** pointer to buffer to store the SIV (16 bytes)
- **out** pointer to buffer to store ciphertext

See:

- wc_AesSivDecrypt_ex
- wc_AesSivEncrypt

Return:

- 0 On success.
- BAD_FUNC_ARG If parameters are invalid.
- Other negative values on error.

Example

```
byte key[32] = { }; // 256-bit key for AES-128-SIV
AesSivAssoc assoc[1];
byte aad[20] = { }; // associated data
byte nonce[12] = { }; // nonce
byte plaintext[100] = { }; // data
byte siv[16];
byte ciphertext[100];

assoc[0].data = aad;
assoc[0].sz = 20;

int ret = wc_AesSivEncrypt_ex(key, 32, assoc, 1, nonce, 12,
                              plaintext, 100, siv, ciphertext);
if (ret != 0) {
    // encryption failed
}
```

```
int wc_AesSivDecrypt_ex(
    const byte * key,
    word32 keySz,
    const AesSivAssoc * assoc,
    word32 numAssoc,
    const byte * nonce,
    word32 nonceSz,
    const byte * in,
    word32 inSz,
    byte * siv,
    byte * out
)
```

This function performs AES-SIV (Synthetic IV) decryption with extended parameters. It verifies the SIV and decrypts the ciphertext.

Parameters:

- **key** pointer to the key buffer (32, 48, or 64 bytes for SIV)
- **keySz** length of the key in bytes
- **assoc** pointer to array of associated data structures
- **numAssoc** number of associated data items
- **nonce** pointer to the nonce buffer (can be NULL)
- **nonceSz** length of the nonce in bytes
- **in** pointer to ciphertext to decrypt
- **inSz** length of ciphertext in bytes
- **siv** pointer to the SIV to verify (16 bytes)
- **out** pointer to buffer to store plaintext

See:

- wc_AesSivEncrypt_ex
- wc_AesSivDecrypt

Return:

- 0 On successful decryption and verification.
- AES_SIV_AUTH_E If SIV verification fails.
- BAD_FUNC_ARG If parameters are invalid.
- Other negative values on error.

Example

```
byte key[32] = { }; // 256-bit key for AES-128-SIV
AesSivAssoc assoc[1];
byte aad[20] = { }; // associated data
byte nonce[12] = { }; // nonce
byte ciphertext[100] = { }; // encrypted data
byte siv[16] = { }; // received SIV
byte plaintext[100];

assoc[0].data = aad;
assoc[0].sz = 20;

int ret = wc_AesSivDecrypt_ex(key, 32, assoc, 1, nonce, 12,
                              ciphertext, 100, siv, plaintext);
if (ret != 0) {
    // decryption or verification failed
}
```

```
int wc_AesCbcDecryptWithKey(
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * key,
    word32 keySz,
    const byte * iv
)
```

Decrypts a cipher from the input buffer in, and places the resulting plain text in the output buffer out using cipher block chaining with AES. This function does not require an AES structure to be initialized. Instead, it takes in a key and an iv (initialization vector) and uses these to initialize an AES object and then decrypt the cipher text.

Parameters:

- **out** pointer to the output buffer in which to store the plain text of the decrypted message
- **in** pointer to the input buffer containing cipher text to be decrypted
- **inSz** size of input message
- **key** 16, 24, or 32 byte secret key for decryption
- **keySz** size of key used for decryption

See:

- wc_AesSetKey
- wc_AesSetIV
- wc_AesCbcEncrypt
- wc_AesCbcDecrypt

Return:

- 0 On successfully decrypting message
- BAD_ALIGN_E Returned on block align error
- BAD_FUNC_ARG Returned if key length is invalid or AES object is null during AesSetIV
- MEMORY_E Returned if WOLFSSL_SMALL_STACK is enabled and XMALLOC fails to instantiate an AES object.

Example

```
int ret = 0;
byte key[] = { some 16, 24, or 32 byte key };
byte iv[]  = { some 16 byte iv };
byte cipher[AES_BLOCK_SIZE * n]; //n being a positive integer making
cipher some multiple of 16 bytes
// fill cipher with cipher text
byte plain [AES_BLOCK_SIZE * n];
if ((ret = wc_AesCbcDecryptWithKey(plain, cipher, AES_BLOCK_SIZE, key,
AES_BLOCK_SIZE, iv)) != 0 ) {
// Decrypt Error
}
```

```
int wc_AesCbcEncryptWithKey(
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * key,
    word32 keySz,
    const byte * iv
)
```

This function encrypts a plaintext message and stores the result in the output buffer. It uses AES encryption with cipher block chaining (CBC) mode. This function does not require an AES structure to be initialized. Instead, it takes in a key and an iv and uses these to encrypt the message.

Parameters:

- **out** pointer to the output buffer in which to store the ciphertext of the encrypted message
- **in** pointer to the input buffer containing plaintext to encrypt
- **inSz** size of input message
- **key** 16, 24, or 32 byte secret key for encryption
- **keySz** size of key used for encryption
- **iv** pointer to the 16 byte initialization vector to use

See:

- wc_AesCbcDecryptWithKey
- wc_AesSetKey
- wc_AesCbcEncrypt

Return:

- 0 On successfully encrypting the message
- BAD_ALIGN_E Returned on block align error
- BAD_FUNC_ARG Returned if key length is invalid
- MEMORY_E Returned if WOLFSSL_SMALL_STACK is enabled and XMALLOC fails to instantiate an AES object

Example

```
byte key[]; // 16, 24, or 32 byte key
byte iv[]; // 16 byte iv
byte plain[]; // plaintext to encrypt
byte cipher[sizeof(plain)];

int ret = wc_AesCbcEncryptWithKey(cipher, plain, sizeof(plain),
                                  key, sizeof(key), iv);
if (ret != 0) {
    // encryption error
}
```

B.13 Algorithms - ARC4

B.12.2.92 function wc_AesCbcEncryptWithKey

B.13.1 Functions

	Name
int	**wc_Arc4Process**(Arc4 * arc4, byte * out, const byte * in, word32 length)This function encrypts an input message from the buffer in, placing the ciphertext in the output buffer out, or decrypts a ciphertext from the buffer in, placing the plaintext in the output buffer out, using ARC4 encryption. This function is used for both encryption and decryption. Before this method may be called, one must first initialize the ARC4 structure using wc_Arc4SetKey.
int	**wc_Arc4SetKey**(Arc4 * arc4, const byte * key, word32 length)This function sets the key for a ARC4 object, initializing it for use as a cipher. It should be called before using it for encryption with wc_Arc4Process.
int	**wc_Arc4Init**(Arc4 * arc4, void * heap, int devId)This function initializes an ARC4 structure for use with asynchronous cryptographic operations. It sets up the heap hint and device ID for hardware acceleration support.
void	**wc_Arc4Free**(Arc4 * arc4)This function frees an ARC4 structure, releasing any resources allocated for asynchronous cryptographic operations. It should be called when the ARC4 structure is no longer needed.

B.13.2 Functions Documentation

```
int wc_Arc4Process(
    Arc4 * arc4,
    byte * out,
    const byte * in,
    word32 length
)
```

This function encrypts an input message from the buffer in, placing the ciphertext in the output buffer out, or decrypts a ciphertext from the buffer in, placing the plaintext in the output buffer out, using ARC4 encryption. This function is used for both encryption and decryption. Before this method may be called, one must first initialize the ARC4 structure using wc_Arc4SetKey.

Parameters:

- **arc4** pointer to the ARC4 structure used to process the message
- **out** pointer to the output buffer in which to store the processed message
- **in** pointer to the input buffer containing the message to process
- **length** length of the message to process

See: wc_Arc4SetKey

Return:

- 0 Returned upon successfully processing the message.
- BAD_FUNC_ARG If arc4, out, or in is NULL.
- MISSING_KEY If no key has been set on the ARC4 structure with wc_Arc4SetKey.

Example

```
Arc4 enc;
byte key[] = { key to use for encryption };
wc_Arc4SetKey(&enc, key, sizeof(key));

byte plain[] = { plain text to encode };
byte cipher[sizeof(plain)];
byte decrypted[sizeof(plain)];
// encrypt the plain into cipher
wc_Arc4Process(&enc, cipher, plain, sizeof(plain));
// decrypt the cipher
wc_Arc4Process(&enc, decrypted, cipher, sizeof(cipher));
```

```
int wc_Arc4SetKey(
    Arc4 * arc4,
    const byte * key,
    word32 length
)
```

This function sets the key for a ARC4 object, initializing it for use as a cipher. It should be called before using it for encryption with wc_Arc4Process.

Parameters:

- **arc4** pointer to an arc4 structure to be used for encryption
- **key** key with which to initialize the arc4 structure
- **length** length of the key used to initialize the arc4 structure

See: wc_Arc4Process

Return: none

Example

```
Arc4 enc;
byte key[] = { initialize with key to use for encryption };
wc_Arc4SetKey(&enc, key, sizeof(key));
```

```
int wc_Arc4Init(
    Arc4 * arc4,
    void * heap,
    int devId
)
```

This function initializes an ARC4 structure for use with asynchronous cryptographic operations. It sets up the heap hint and device ID for hardware acceleration support.

Parameters:

- **arc4** pointer to the Arc4 structure to initialize
- **heap** pointer to heap hint for memory allocation (can be NULL)
- **devId** device ID for hardware acceleration (use INVALID_DEVID for software)

See:

- wc_Arc4SetKey
- wc_Arc4Free

Return:

- 0 On success.
- BAD_FUNC_ARG If arc4 is NULL.

Example

```
Arc4 arc4;
int ret = wc_Arc4Init(&arc4, NULL, INVALID_DEVID);
if (ret != 0) {
    // initialization failed
}
// use arc4 for encryption/decryption
wc_Arc4Free(&arc4);
```

```
void wc_Arc4Free(
    Arc4 * arc4
)
```

This function frees an ARC4 structure, releasing any resources allocated for asynchronous cryptographic operations. It should be called when the ARC4 structure is no longer needed.

Parameters:

- **arc4** pointer to the Arc4 structure to free

See:

- wc_Arc4Init
- wc_Arc4SetKey

Return: none No return value.

Example

```
Arc4 arc4;
wc_Arc4Init(&arc4, NULL, INVALID_DEVID);
wc_Arc4SetKey(&arc4, key, keyLen);
// use arc4 for encryption/decryption
wc_Arc4Free(&arc4);
```

B.14 Algorithms - BLAKE2

B.13.2.4 function wc_Arc4Free

B.14.1 Functions

	Name
int	**wc_InitBlake2b**(Blake2b * b2b, word32 digestSz)This function initializes a Blake2b structure for use with the Blake2 hash function.
int	**wc_Blake2bUpdate**(Blake2b * b2b, const byte * data, word32 sz)This function updates the Blake2b hash with the given input data. This function should be called after wc_InitBlake2b, and repeated until one is ready for the final hash: wc_Blake2bFinal.
int	**wc_Blake2bFinal**(Blake2b * b2b, byte * final, word32 requestSz)This function computes the Blake2b hash of the previously supplied input data. The output hash will be of length requestSz, or, if requestSz==0, the digestSz of the b2b structure. This function should be called after wc_InitBlake2b and wc_Blake2bUpdate has been processed for each piece of input data desired.
int	**wc_Blake2bHmacInit**(Blake2b * b2b, const byte * key, size_t key_len)Initialize an HMAC-BLAKE2b message authentication code computation.
int	**wc_Blake2bHmacUpdate**(Blake2b * b2b, const byte * in, size_t in_len)Update an HMAC-BLAKE2b message authentication code computation with additional input data.
int	**wc_Blake2bHmacFinal**(Blake2b * b2b, const byte * key, size_t key_len, byte * out, size_t out_len)Finalize an HMAC-BLAKE2b message authentication code computation.
int	**wc_Blake2bHmac**(const byte * in, size_t in_len, const byte * key, size_t key_len, byte * out, size_t out_len)Compute the HMAC-BLAKE2b message authentication code of the given input data using the given key.
int	**wc_InitBlake2s**(Blake2s * b2s, word32 digestSz)This function initializes a Blake2s structure for use with the Blake2 hash function.

	Name
int	**wc_Blake2sUpdate**(Blake2s * b2s, const byte * data, word32 sz)This function updates the Blake2s hash with the given input data. This function should be called after wc_InitBlake2s, and repeated until one is ready for the final hash: wc_Blake2sFinal.
int	**wc_Blake2sFinal**(Blake2s * b2s, byte * final, word32 requestSz)This function computes the Blake2s hash of the previously supplied input data. The output hash will be of length requestSz, or, if requestSz==0, the digestSz of the b2s structure. This function should be called after wc_InitBlake2s and wc_Blake2sUpdate has been processed for each piece of input data desired.
int	**wc_Blake2sHmacInit**(Blake2s * b2s, const byte * key, size_t key_len)Initialize an HMAC-BLAKE2s message authentication code computation.
int	**wc_Blake2sHmacUpdate**(Blake2s * b2s, const byte * in, size_t in_len)Update an HMAC-BLAKE2s message authentication code computation with additional input data.
int	**wc_Blake2sHmacFinal**(Blake2s * b2s, const byte * key, size_t key_len, byte * out, size_t out_len)Finalize an HMAC-BLAKE2s message authentication code computation.
int	**wc_Blake2sHmac**(const byte * in, size_t in_len, const byte * key, size_t key_len, byte * out, size_t out_len)This function computes the HMAC-BLAKE2s message authentication code of the given input data using the given key.

B.14.2 Functions Documentation

```
int wc_InitBlake2b(
    Blake2b * b2b,
    word32 digestSz
)
```

This function initializes a Blake2b structure for use with the Blake2 hash function.

Parameters:

- **b2b** pointer to the Blake2b structure to initialize
- **digestSz** length of the blake 2 digest to implement

See: wc_Blake2bUpdate

Return: 0 Returned upon successfully initializing the Blake2b structure and setting the digest size.

Example

```
Blake2b b2b;
// initialize Blake2b structure with 64 byte digest
wc_InitBlake2b(&b2b, WC_BLAKE2B_DIGEST_SIZE);
```

```
int wc_Blake2bUpdate(
    Blake2b * b2b,
    const byte * data,
    word32 sz
)
```

This function updates the Blake2b hash with the given input data. This function should be called after wc_InitBlake2b, and repeated until one is ready for the final hash: wc_Blake2bFinal.

Parameters:

- **b2b** pointer to the Blake2b structure to update
- **data** pointer to a buffer containing the data to append
- **sz** length of the input data to append

See:

- wc_InitBlake2b
- wc_Blake2bFinal

Return:

- 0 Returned upon successfully update the Blake2b structure with the given data
- -1 Returned if there is a failure while compressing the input data

Example

```
int ret;
Blake2b b2b;
// initialize Blake2b structure with 64 byte digest
wc_InitBlake2b(&b2b, WC_BLAKE2B_DIGEST_SIZE);

byte plain[] = { // initialize input };

ret = wc_Blake2bUpdate(&b2b, plain, sizeof(plain));
if (ret != 0) {
    // error updating blake2b
}
```

```
int wc_Blake2bFinal(
    Blake2b * b2b,
    byte * final,
    word32 requestSz
)
```

This function computes the Blake2b hash of the previously supplied input data. The output hash will be of length requestSz, or, if requestSz==0, the digestSz of the b2b structure. This function should be called after wc_InitBlake2b and wc_Blake2bUpdate has been processed for each piece of input data desired.

Parameters:

- **b2b** pointer to the Blake2b structure to update
- **final** pointer to a buffer in which to store the blake2b hash. Should be of length requestSz
- **requestSz** length of the digest to compute. When this is zero, b2b->digestSz will be used instead

See:

- wc_InitBlake2b
- wc_Blake2bUpdate

Return:

- 0 Returned upon successfully computing the Blake2b hash
- -1 Returned if there is a failure while parsing the Blake2b hash

Example

```
int ret;
Blake2b b2b;
byte hash[WC_BLAKE2B_DIGEST_SIZE];
// initialize Blake2b structure with 64 byte digest
wc_InitBlake2b(&b2b, WC_BLAKE2B_DIGEST_SIZE);
... // call wc_Blake2bUpdate to add data to hash

ret = wc_Blake2bFinal(&b2b, hash, WC_BLAKE2B_DIGEST_SIZE);
if (ret != 0) {
    // error generating blake2b hash
}
```

```
int wc_Blake2bHmacInit(
    Blake2b * b2b,
    const byte * key,
    size_t key_len
)
```

Initialize an HMAC-BLAKE2b message authentication code computation.

Parameters:

- **b2b** Blake2b structure to be used for the MAC computation.
- **key** pointer to the key
- **key_len** length of the key

Return: 0 Returned upon successfully initializing the HMAC-BLAKE2b MAC computation.

Example

```
Blake2b b2b;
int ret;
byte key[] = {4, 5, 6};
ret = wc_Blake2bHmacInit(&b2b, key);
if (ret != 0) {
    // error generating HMAC-BLAKE2b
}
```

```
int wc_Blake2bHmacUpdate(
    Blake2b * b2b,
    const byte * in,
    size_t in_len
)
```

Update an HMAC-BLAKE2b message authentication code computation with additional input data.

Parameters:

- **b2b** Blake2b structure to be used for the MAC computation.
- **in** pointer to the input data
- **in_len** length of the input data

Return: 0 Returned upon successfully updating the HMAC-BLAKE2b MAC computation.

Example

```
Blake2b b2b;
int ret;
byte key[] = {4, 5, 6};
byte data[] = {1, 2, 3};
ret = wc_Blake2bHmacInit(&b2b, key, sizeof(key));
ret = wc_Blake2bHmacUpdate(&b2b, data, sizeof(data));
```

```
int wc_Blake2bHmacFinal(
    Blake2b * b2b,
    const byte * key,
    size_t key_len,
    byte * out,
    size_t out_len
)
```

Finalize an HMAC-BLAKE2b message authentication code computation.

Parameters:

- **b2b** Blake2b structure to be used for the MAC computation.
- **key** pointer to the key
- **key_len** length of the key
- **out** output buffer to store computed MAC
- **out_len** length of output buffer

Return: 0 Returned upon successfully finalizing the HMAC-BLAKE2b MAC computation.

Example

```
Blake2b b2b;
int ret;
byte key[] = {4, 5, 6};
byte data[] = {1, 2, 3};
byte mac[WC_BLAKE2B_DIGEST_SIZE];
ret = wc_Blake2bHmacInit(&b2b, key, sizeof(key));
ret = wc_Blake2bHmacUpdate(&b2b, data, sizeof(data));
ret = wc_Blake2bHmacFinalize(&b2b, key, sizeof(key), mac, sizezof(mac));
```

```
int wc_Blake2bHmac(
    const byte * in,
    size_t in_len,
    const byte * key,
    size_t key_len,
    byte * out,
    size_t out_len
)
```

Compute the HMAC-BLAKE2b message authentication code of the given input data using the given key.

Parameters:

- **in** pointer to the input data
- **in_len** length of the input data
- **key** pointer to the key
- **key_len** length of the key
- **out** output buffer to store computed MAC
- **out_len** length of output buffer

Return: 0 Returned upon successfully computing the HMAC-BLAKE2b MAC.

Example

```
int ret;
byte mac[WC_BLAKE2B_DIGEST_SIZE];
byte data[] = {1, 2, 3};
byte key[] = {4, 5, 6};
ret = wc_Blake2bHmac(data, sizeof(data), key, sizeof(key), mac, sizeof(mac));
if (ret != 0) {
    // error generating HMAC-BLAKE2b
}
```

```
int wc_InitBlake2s(
    Blake2s * b2s,
    word32 digestSz
)
```

This function initializes a Blake2s structure for use with the Blake2 hash function.

Parameters:

- **b2s** pointer to the Blake2s structure to initialize
- **digestSz** length of the blake 2 digest to implement

See: wc_Blake2sUpdate

Return: 0 Returned upon successfully initializing the Blake2s structure and setting the digest size.

Example

```
Blake2s b2s;
// initialize Blake2s structure with 32 byte digest
wc_InitBlake2s(&b2s, WC_BLAKE2S_DIGEST_SIZE);
```

```
int wc_Blake2sUpdate(
    Blake2s * b2s,
    const byte * data,
    word32 sz
)
```

This function updates the Blake2s hash with the given input data. This function should be called after wc_InitBlake2s, and repeated until one is ready for the final hash: wc_Blake2sFinal.

Parameters:

- **b2s** pointer to the Blake2s structure to update
- **data** pointer to a buffer containing the data to append
- **sz** length of the input data to append

See:

- wc_InitBlake2s
- wc_Blake2sFinal

Return:

- 0 Returned upon successfully update the Blake2s structure with the given data
- -1 Returned if there is a failure while compressing the input data

Example

```
int ret;
Blake2s b2s;
// initialize Blake2s structure with 32 byte digest
wc_InitBlake2s(&b2s, WC_BLAKE2S_DIGEST_SIZE);

byte plain[] = { // initialize input };

ret = wc_Blake2sUpdate(&b2s, plain, sizeof(plain));
if (ret != 0) {
    // error updating blake2s
}
```

```
int wc_Blake2sFinal(
    Blake2s * b2s,
    byte * final,
    word32 requestSz
)
```

This function computes the Blake2s hash of the previously supplied input data. The output hash will be of length requestSz, or, if requestSz==0, the digestSz of the b2s structure. This function should be called after wc_InitBlake2s and wc_Blake2sUpdate has been processed for each piece of input data desired.

Parameters:

- **b2s** pointer to the Blake2s structure to update
- **final** pointer to a buffer in which to store the blake2s hash. Should be of length requestSz
- **requestSz** length of the digest to compute. When this is zero, b2s->digestSz will be used instead

See:

- wc_InitBlake2s
- wc_Blake2sUpdate

Return:

- 0 Returned upon successfully computing the Blake2s hash
- -1 Returned if there is a failure while parsing the Blake2s hash

Example

```
int ret;
Blake2s b2s;
byte hash[WC_BLAKE2S_DIGEST_SIZE];
// initialize Blake2s structure with 32 byte digest
wc_InitBlake2s(&b2s, WC_BLAKE2S_DIGEST_SIZE);
... // call wc_Blake2sUpdate to add data to hash

ret = wc_Blake2sFinal(&b2s, hash, WC_BLAKE2S_DIGEST_SIZE);
if (ret != 0) {
```

```
    // error generating blake2s hash
}
```

```
int wc_Blake2sHmacInit(
    Blake2s * b2s,
    const byte * key,
    size_t key_len
)
```

Initialize an HMAC-BLAKE2s message authentication code computation.

Parameters:

- **b2s** Blake2s structure to be used for the MAC computation.
- **key** pointer to the key
- **key_len** length of the key

Return: 0 Returned upon successfully initializing the HMAC-BLAKE2s MAC computation.

Example

```
Blake2s b2s;
int ret;
byte key[] = {4, 5, 6};
ret = wc_Blake2sHmacInit(&b2s, key);
if (ret != 0) {
    // error generating HMAC-BLAKE2s
}
```

```
int wc_Blake2sHmacUpdate(
    Blake2s * b2s,
    const byte * in,
    size_t in_len
)
```

Update an HMAC-BLAKE2s message authentication code computation with additional input data.

Parameters:

- **b2s** Blake2s structure to be used for the MAC computation.
- **in** pointer to the input data
- **in_len** length of the input data

Return: 0 Returned upon successfully updating the HMAC-BLAKE2s MAC computation.

Example

```
Blake2s b2s;
int ret;
byte key[] = {4, 5, 6};
byte data[] = {1, 2, 3};
ret = wc_Blake2sHmacInit(&b2s, key, sizeof(key));
ret = wc_Blake2sHmacUpdate(&b2s, data, sizeof(data));
```

```
int wc_Blake2sHmacFinal(
    Blake2s * b2s,
    const byte * key,
    size_t key_len,
    byte * out,
    size_t out_len
)
```

Finalize an HMAC-BLAKE2s message authentication code computation.

Parameters:

- **b2s** Blake2s structure to be used for the MAC computation.
- **key** pointer to the key
- **key_len** length of the key
- **out** output buffer to store computed MAC
- **out_len** length of output buffer

Return: 0 Returned upon successfully finalizing the HMAC-BLAKE2s MAC computation.

Example

```
Blake2s b2s;
int ret;
byte key[] = {4, 5, 6};
byte data[] = {1, 2, 3};
byte mac[WC_BLAKE2S_DIGEST_SIZE];
ret = wc_Blake2sHmacInit(&b2s, key, sizeof(key));
ret = wc_Blake2sHmacUpdate(&b2s, data, sizeof(data));
ret = wc_Blake2sHmacFinalize(&b2s, key, sizeof(key), mac, sizezof(mac));
```

```
int wc_Blake2sHmac(
    const byte * in,
    size_t in_len,
    const byte * key,
    size_t key_len,
    byte * out,
    size_t out_len
)
```

This function computes the HMAC-BLAKE2s message authentication code of the given input data using the given key.

Parameters:

- **in** pointer to the input data
- **in_len** length of the input data
- **key** pointer to the key
- **key_len** length of the key
- **out** output buffer to store computed MAC
- **out_len** length of output buffer

Return: 0 Returned upon successfully computing the HMAC-BLAKE2s MAC.

Example

```
int ret;
byte mac[WC_BLAKE2S_DIGEST_SIZE];
byte data[] = {1, 2, 3};
byte key[] = {4, 5, 6};
ret = wc_Blake2sHmac(data, sizeof(data), key, sizeof(key), mac, sizeof(mac));
if (ret != 0) {
    // error generating HMAC-BLAKE2s
}
```

B.15 Algorithms - Camellia

B.14.2.14 function wc_Blake2sHmac

B.15.1 Functions

	Name
int	**wc_CamelliaSetKey**(wc_Camellia * cam, const byte * key, word32 len, const byte * iv)This function sets the key and initialization vector for a camellia object, initializing it for use as a cipher.
int	**wc_CamelliaSetIV**(wc_Camellia * cam, const byte * iv)This function sets the initialization vector for a camellia object.
int	**wc_CamelliaEncryptDirect**(wc_Camellia * cam, byte * out, const byte * in)This function does a one-block encrypt using the provided camellia object. It parses the first 16 byte block from the buffer in and stores the encrypted result in the buffer out. Before using this function, one should initialize the camellia object using wc_CamelliaSetKey.

	Name
int	**wc_CamelliaDecryptDirect**(wc_Camellia * cam, byte * out, const byte * in)This function does a one-block decrypt using the provided camellia object. It parses the first 16 byte block from the buffer in, decrypts it, and stores the result in the buffer out. Before using this function, one should initialize the camellia object using wc_CamelliaSetKey.
int	**wc_CamelliaCbcEncrypt**(wc_Camellia * cam, byte * out, const byte * in, word32 sz)This function encrypts the plaintext from the buffer in and stores the output in the buffer out. It performs this encryption using Camellia with Cipher Block Chaining (CBC).
int	**wc_CamelliaCbcDecrypt**(wc_Camellia * cam, byte * out, const byte * in, word32 sz)This function decrypts the ciphertext from the buffer in and stores the output in the buffer out. It performs this decryption using Camellia with Cipher Block Chaining (CBC).

B.15.2 Functions Documentation

```
int wc_CamelliaSetKey(
    wc_Camellia * cam,
    const byte * key,
    word32 len,
    const byte * iv
)
```

This function sets the key and initialization vector for a camellia object, initializing it for use as a cipher.

Parameters:

- **cam** pointer to the camellia structure on which to set the key and iv
- **key** pointer to the buffer containing the 16, 24, or 32 byte key to use for encryption and decryption
- **len** length of the key passed in
- **iv** pointer to the buffer containing the 16 byte initialization vector for use with this camellia structure

See:

- wc_CamelliaEncryptDirect
- wc_CamelliaDecryptDirect
- wc_CamelliaCbcEncrypt
- wc_CamelliaCbcDecrypt

Return:

- 0 Returned upon successfully setting the key and initialization vector
- BAD_FUNC_ARG returned if there is an error processing one of the input arguments
- MEMORY_E returned if there is an error allocating memory with XMALLOC

Example

```
Camellia cam;
byte key[32];
// initialize key
byte iv[16];
// initialize iv
if( wc_CamelliaSetKey(&cam, key, sizeof(key), iv) != 0) {
    // error initializing camellia structure
}
```

```
int wc_CamelliaSetIV(
    wc_Camellia * cam,
    const byte * iv
)
```

This function sets the initialization vector for a camellia object.

Parameters:

- **cam** pointer to the camellia structure on which to set the iv
- **iv** pointer to the buffer containing the 16 byte initialization vector for use with this camellia structure

See: wc_CamelliaSetKey

Return:

- 0 Returned upon successfully setting the key and initialization vector
- BAD_FUNC_ARG returned if there is an error processing one of the input arguments

Example

```
Camellia cam;
byte iv[16];
// initialize iv
if( wc_CamelliaSetIV(&cam, iv) != 0) {
// error initializing camellia structure
}
```

```
int wc_CamelliaEncryptDirect(
    wc_Camellia * cam,
    byte * out,
    const byte * in
)
```

This function does a one-block encrypt using the provided camellia object. It parses the first 16 byte block from the buffer in and stores the encrypted result in the buffer out. Before using this function, one should initialize the camellia object using wc_CamelliaSetKey.

Parameters:

- **cam** pointer to the camellia structure to use for encryption
- **out** pointer to the buffer in which to store the encrypted block
- **in** pointer to the buffer containing the plaintext block to encrypt

See: wc_CamelliaDecryptDirect

Return: none No returns.

Example

```
Camellia cam;
// initialize cam structure with key and iv
byte plain[] = { // initialize with message to encrypt };
byte cipher[16];

wc_CamelliaEncryptDirect(&ca, cipher, plain);
```

```
int wc_CamelliaDecryptDirect(
    wc_Camellia * cam,
    byte * out,
    const byte * in
)
```

This function does a one-block decrypt using the provided camellia object. It parses the first 16 byte block from the buffer in, decrypts it, and stores the result in the buffer out. Before using this function, one should initialize the camellia object using wc_CamelliaSetKey.

Parameters:

- **cam** pointer to the camellia structure to use for encryption
- **out** pointer to the buffer in which to store the decrypted plaintext block
- **in** pointer to the buffer containing the ciphertext block to decrypt

See: wc_CamelliaEncryptDirect

Return: none No returns.

Example

```
Camellia cam;
// initialize cam structure with key and iv
byte cipher[] = { // initialize with encrypted message to decrypt };
byte decrypted[16];

wc_CamelliaDecryptDirect(&cam, decrypted, cipher);
```

```
int wc_CamelliaCbcEncrypt(
    wc_Camellia * cam,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function encrypts the plaintext from the buffer in and stores the output in the buffer out. It performs this encryption using Camellia with Cipher Block Chaining (CBC).

Parameters:

- **cam** pointer to the camellia structure to use for encryption
- **out** pointer to the buffer in which to store the encrypted ciphertext
- **in** pointer to the buffer containing the plaintext to encrypt
- **sz** the size of the message to encrypt

See: wc_CamelliaCbcDecrypt

Return: none No returns.

Example

```
Camellia cam;
// initialize cam structure with key and iv
byte plain[] = { // initialize with encrypted message to decrypt };
byte cipher[sizeof(plain)];

wc_CamelliaCbcEncrypt(&cam, cipher, plain, sizeof(plain));
```

```
int wc_CamelliaCbcDecrypt(
    wc_Camellia * cam,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function decrypts the ciphertext from the buffer in and stores the output in the buffer out. It performs this decryption using Camellia with Cipher Block Chaining (CBC).

Parameters:

- **cam** pointer to the camellia structure to use for encryption
- **out** pointer to the buffer in which to store the decrypted message
- **in** pointer to the buffer containing the encrypted ciphertext
- **sz** the size of the message to encrypt

See: wc_CamelliaCbcEncrypt

Return: none No returns.

Example

```
Camellia cam;
// initialize cam structure with key and iv
byte cipher[] = { // initialize with encrypted message to decrypt };
byte decrypted[sizeof(cipher)];

wc_CamelliaCbcDecrypt(&cam, decrypted, cipher, sizeof(cipher));
```

B.16 Algorithms - ChaCha

B.15.2.6 function wc_CamelliaCbcDecrypt

B.16.1 Functions

	Name
int	**wc_Chacha_SetIV**(ChaCha * ctx, const byte * inIv, word32 counter)This function sets the initialization vector (nonce) for a ChaCha object, initializing it for use as a cipher. It should be called after the key has been set, using wc_Chacha_SetKey. A difference nonce should be used for each round of encryption.
int	**wc_Chacha_Process**(ChaCha * ctx, byte * cipher, const byte * plain, word32 msglen)This function processes the text from the buffer input, encrypts or decrypts it, and stores the result in the buffer output.
int	**wc_Chacha_SetKey**(ChaCha * ctx, const byte * key, word32 keySz)This function sets the key for a ChaCha object, initializing it for use as a cipher. It should be called before setting the nonce with wc_Chacha_SetIV, and before using it for encryption with wc_Chacha_Process.
int	**wc_XChacha_SetKey**(ChaCha * ctx, const byte * key, word32 keySz, const byte * nonce, word32 nonceSz, word32 counter)This function sets the key and nonce for an XChaCha cipher context. XChaCha extends ChaCha20 to use a 192-bit nonce instead of 96 bits, providing better security for applications that need to encrypt many messages with the same key.

B.16.2 Functions Documentation

```
int wc_Chacha_SetIV(
    ChaCha * ctx,
    const byte * inIv,
    word32 counter
)
```

This function sets the initialization vector (nonce) for a ChaCha object, initializing it for use as a cipher. It should be called after the key has been set, using wc_Chacha_SetKey. A difference nonce should be used for each round of encryption.

Parameters:

- **ctx** pointer to the ChaCha structure on which to set the iv
- **inIv** pointer to a buffer containing the 12 byte initialization vector with which to initialize the ChaCha structure
- **counter** the value at which the block counter should start–usually zero.

See:

- wc_Chacha_SetKey
- wc_Chacha_Process

Return:

- 0 Returned upon successfully setting the initialization vector
- BAD_FUNC_ARG returned if there is an error processing the ctx input argument

Example

```
ChaCha enc;
// initialize enc with wc_Chacha_SetKey
byte iv[12];
// initialize iv
if( wc_Chacha_SetIV(&enc, iv, 0) != 0) {
    // error initializing ChaCha structure
}
```

```
int wc_Chacha_Process(
    ChaCha * ctx,
    byte * cipher,
    const byte * plain,
    word32 msglen
)
```

This function processes the text from the buffer input, encrypts or decrypts it, and stores the result in the buffer output.

Parameters:

- **ctx** pointer to the ChaCha structure on which to set the iv
- **output** pointer to a buffer in which to store the output ciphertext or decrypted plaintext
- **input** pointer to the buffer containing the input plaintext to encrypt or the input ciphertext to decrypt
- **msglen** length of the message to encrypt or the ciphertext to decrypt

See:

- wc_Chacha_SetKey
- wc_Chacha_Process

Return:

- 0 Returned upon successfully encrypting or decrypting the input
- BAD_FUNC_ARG returned if there is an error processing the ctx input argument
- MISSING_KEY returned if wc_Chacha_SetKey was not called to set a key before processing

Example

```
ChaCha enc;
// initialize enc with wc_Chacha_SetKey and wc_Chacha_SetIV

byte plain[] = { // initialize plaintext };
byte cipher[sizeof(plain)];
if( wc_Chacha_Process(&enc, cipher, plain, sizeof(plain)) != 0) {
    // error processing ChaCha cipher
}
```

```
int wc_Chacha_SetKey(
    ChaCha * ctx,
    const byte * key,
    word32 keySz
)
```

This function sets the key for a ChaCha object, initializing it for use as a cipher. It should be called before setting the nonce with wc_Chacha_SetIV, and before using it for encryption with wc_Chacha_Process.

Parameters:

- **ctx** pointer to the ChaCha structure in which to set the key
- **key** pointer to a buffer containing the 16 or 32 byte key with which to initialize the ChaCha structure
- **keySz** the length of the key passed in

See:

- wc_Chacha_SetIV
- wc_Chacha_Process

Return:

- 0 Returned upon successfully setting the key
- BAD_FUNC_ARG returned if there is an error processing the ctx input argument or if the key is not 16 or 32 bytes long

Example

```
ChaCha enc;
byte key[] = { // initialize key };

if( wc_Chacha_SetKey(&enc, key, sizeof(key)) != 0) {
    // error initializing ChaCha structure
}
```

```
int wc_XChacha_SetKey(
    ChaCha * ctx,
    const byte * key,
    word32 keySz,
    const byte * nonce,
    word32 nonceSz,
    word32 counter
)
```

This function sets the key and nonce for an XChaCha cipher context. XChaCha extends ChaCha20 to use a 192-bit nonce instead of 96 bits, providing better security for applications that need to encrypt many messages with the same key.

Parameters:

- **ctx** pointer to the ChaCha structure to initialize
- **key** pointer to the key buffer (16 or 32 bytes)
- **keySz** length of the key in bytes (16 or 32)
- **nonce** pointer to the nonce buffer (must be 24 bytes)
- **nonceSz** length of the nonce in bytes (must be 24)
- **counter** initial block counter value (usually 0)

See:

- wc_Chacha_SetKey
- wc_Chacha_SetIV
- wc_Chacha_Process

Return:

- 0 On success.
- BAD_FUNC_ARG If ctx, key, or nonce is NULL, or if keySz is invalid, or if nonceSz is not XCHACHA_NONCE_BYTES (24 bytes).
- Other negative values on error.

Example

```
ChaCha ctx;
byte key[32] = { }; // 256-bit key
byte nonce[24] = { }; // 192-bit nonce
```

```
byte plaintext[100] = { }; // data to encrypt
byte ciphertext[100];

int ret = wc_XChacha_SetKey(&ctx, key, 32, nonce, 24, 0);
if (ret != 0) {
    // error setting XChaCha key
}
wc_Chacha_Process(&ctx, ciphertext, plaintext, 100);
```

B.17 Algorithms - ChaCha20_Poly1305

B.16.2.4 function wc_XChacha_SetKey

B.17.1 Functions

	Name
int	**wc_ChaCha20Poly1305_Encrypt**(const byte inKey[CHACHA20_POLY1305_AEAD_KEYSIZE], const byte inIV[CHACHA20_POLY1305_AEAD_IV_SIZE], const byte * inAAD, word32 inAADLen, const byte * inPlaintext, word32 inPlaintextLen, byte * outCiphertext, byte outAuth-Tag[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE])This function encrypts an input message, inPlaintext, using the ChaCha20 stream cipher, into the output buffer, outCiphertext. It also performs Poly_1305 authentication (on the cipher text), and stores the generated authentication tag in the output buffer, outAuthTag.
int	**wc_ChaCha20Poly1305_Decrypt**(const byte inKey[CHACHA20_POLY1305_AEAD_KEYSIZE], const byte inIV[CHACHA20_POLY1305_AEAD_IV_SIZE], const byte * inAAD, word32 inAADLen, const byte * inCiphertext, word32 inCiphertextLen, const byte inAuth-Tag[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE], byte * outPlaintext)This function decrypts input ciphertext, inCiphertext, using the ChaCha20 stream cipher, into the output buffer, outPlaintext. It also performs Poly_1305 authentication, comparing the given inAuthTag to an authentication generated with the inAAD (arbitrary length additional authentication data). If a nonzero error code is returned, the output data, outPlaintext, is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.

	Name
int	**wc_ChaCha20Poly1305_Encrypt_ex**(ChaCha * chacha, Poly1305 * poly, byte * out, const byte * in, word32 sz, const byte * nonce, byte * tag, const byte * aad, word32 aadSz)This function performs the same AEAD encryption as wc_ChaCha20Poly1305_Encrypt, but takes a caller_owned ChaCha20 context whose key has already been set (with wc_Chacha_SetKey) plus a Poly1305 context, instead of a raw key. This lets the ChaCha20 key be set once and reused across many records, varying only the per-record nonce - it is intended for protocol record layers such as TLS. A fresh per-record Poly1305 key is derived from the ChaCha20 keystream, so this call re-keys the supplied Poly1305 context; the caller does not key it. The 16 byte authentication tag over the AAD and ciphertext is written to tag.
int	**wc_ChaCha20Poly1305_Decrypt_ex**(ChaCha * chacha, Poly1305 * poly, byte * out, const byte * in, word32 sz, const byte * nonce, const byte * tag, const byte * aad, word32 aadSz)This function is the decryption counterpart of wc_ChaCha20Poly1305_Encrypt_ex. It takes a caller_owned ChaCha20 context whose key has already been set (with wc_Chacha_SetKey) plus a Poly1305 context, decrypts in to out, and verifies the Poly1305 tag over the AAD and ciphertext. On tag mismatch it returns MAC_CMP_FAILED_E and zeroizes the output buffer, so no unauthenticated plaintext is released. The ChaCha20 key is reused across records, varying only the per_record nonce; the Poly1305 context is re_keyed internally on each call. out may alias in (in_place decryption is supported).
int	**wc_ChaCha20Poly1305_CheckTag**(const byte authTag[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE], const byte authTagChk[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE])Compares two authentication tags in constant time to prevent timing attacks.
int	**wc_ChaCha20Poly1305_Init**(ChaChaPoly_Aead * aead, const byte inKey[CHACHA20_POLY1305_AEAD_KEYSIZE], const byte inIV[CHACHA20_POLY1305_AEAD_IV_SIZE], int isEncrypt)Initializes a ChaChaPoly_Aead structure for incremental encryption or decryption operations.

	Name
int	**wc_ChaCha20Poly1305_UpdateAad**(ChaChaPoly_Aead * aead, const byte * inAAD, word32 inAADLen)Updates the AEAD context with additional authenticated data (AAD). Must be called after Init and before UpdateData.
int	**wc_ChaCha20Poly1305_UpdateData**(ChaChaPoly_Aead * aead, const byte * inData, byte * outData, word32 dataLen)Encrypts or decrypts data incrementally. Can be called multiple times to process data in chunks.
int	**wc_ChaCha20Poly1305_Final**(ChaChaPoly_Aead * aead, byte outAuth-Tag[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE])Finalizes the AEAD operation and generates the authentication tag.
int	**wc_XChaCha20Poly1305_Init**(ChaChaPoly_Aead * aead, const byte * ad, word32 ad_len, const byte * inKey, word32 inKeySz, const byte * inIV, word32 inIVSz, int isEncrypt)Initializes XChaCha20-Poly1305 AEAD with extended nonce. XChaCha20 uses a 24-byte nonce instead of 12-byte.
int	**wc_XChaCha20Poly1305_Encrypt**(byte * dst, size_t dst_space, const byte * src, size_t src_len, const byte * ad, size_t ad_len, const byte * nonce, size_t nonce_len, const byte * key, size_t key_len)One-shot XChaCha20-Poly1305 encryption with 24-byte nonce.
int	**wc_XChaCha20Poly1305_Decrypt**(byte * dst, size_t dst_space, const byte * src, size_t src_len, const byte * ad, size_t ad_len, const byte * nonce, size_t nonce_len, const byte * key, size_t key_len)One-shot XChaCha20-Poly1305 decryption with 24-byte nonce.

B.17.2 Functions Documentation

```
int wc_ChaCha20Poly1305_Encrypt(
    const byte inKey[CHACHA20_POLY1305_AEAD_KEYSIZE],
    const byte inIV[CHACHA20_POLY1305_AEAD_IV_SIZE],
    const byte * inAAD,
    word32 inAADLen,
    const byte * inPlaintext,
    word32 inPlaintextLen,
    byte * outCiphertext,
    byte outAuthTag[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE]
)
```

This function encrypts an input message, inPlaintext, using the ChaCha20 stream cipher, into the output buffer, outCiphertext. It also performs Poly-1305 authentication (on the cipher text), and stores the generated authentication tag in the output buffer, outAuthTag.

Parameters:

- **inKey** pointer to a buffer containing the 32 byte key to use for encryption
- **inIv** pointer to a buffer containing the 12 byte iv to use for encryption
- **inAAD** pointer to the buffer containing arbitrary length additional authenticated data (AAD)
- **inAADLen** length of the input AAD
- **inPlaintext** pointer to the buffer containing the plaintext to encrypt
- **inPlaintextLen** the length of the plain text to encrypt
- **outCiphertext** pointer to the buffer in which to store the ciphertext
- **outAuthTag** pointer to a 16 byte wide buffer in which to store the authentication tag

See:

- wc_ChaCha20Poly1305_Decrypt
- wc_ChaCha_*
- wc_Poly1305*

Return:

- 0 Returned upon successfully encrypting the message
- BAD_FUNC_ARG returned if there is an error during the encryption process

Example

```
byte key[] = { // initialize 32 byte key };
byte iv[]  = { // initialize 12 byte key };
byte inAAD[] = { // initialize AAD };

byte plain[] = { // initialize message to encrypt };
byte cipher[sizeof(plain)];
byte authTag[16];

int ret = wc_ChaCha20Poly1305_Encrypt(key, iv, inAAD, sizeof(inAAD),
plain, sizeof(plain), cipher, authTag);

if(ret != 0) {
    // error running encrypt
}
```

```
int wc_ChaCha20Poly1305_Decrypt(
    const byte inKey[CHACHA20_POLY1305_AEAD_KEYSIZE],
    const byte inIV[CHACHA20_POLY1305_AEAD_IV_SIZE],
    const byte * inAAD,
    word32 inAADLen,
    const byte * inCiphertext,
    word32 inCiphertextLen,
    const byte inAuthTag[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE],
    byte * outPlaintext
)
```

This function decrypts input ciphertext, inCiphertext, using the ChaCha20 stream cipher, into the output buffer, outPlaintext. It also performs Poly-1305 authentication, comparing the given inAuthTag to an authentication generated with the inAAD (arbitrary length additional authentication data). If a nonzero error code is returned, the output data, outPlaintext, is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.

Parameters:

- **inKey** pointer to a buffer containing the 32 byte key to use for decryption
- **inIv** pointer to a buffer containing the 12 byte iv to use for decryption
- **inAAD** pointer to the buffer containing arbitrary length additional authenticated data (AAD)
- **inAADLen** length of the input AAD
- **inCiphertext** pointer to the buffer containing the ciphertext to decrypt
- **outCiphertextLen** the length of the ciphertext to decrypt
- **inAuthTag** pointer to the buffer containing the 16 byte digest for authentication
- **outPlaintext** pointer to the buffer in which to store the plaintext

See:

- wc_ChaCha20Poly1305_Encrypt
- wc_ChaCha_*
- wc_Poly1305*

Return:

- 0 Returned upon successfully decrypting and authenticating the message
- BAD_FUNC_ARG Returned if any of the function arguments do not match what is expected
- MAC_CMP_FAILED_E Returned if the generated authentication tag does not match the supplied inAuthTag.
- MEMORY_E Returned if internal buffer allocation failed.
- CHACHA_POLY_OVERFLOW Can be returned if input is corrupted.

Example

```
byte key[]   = { // initialize 32 byte key };
byte iv[]    = { // initialize 12 byte key };
byte inAAD[] = { // initialize AAD };

byte cipher[]    = { // initialize with received ciphertext };
byte authTag[16] = { // initialize with received authentication tag };

byte plain[sizeof(cipher)];

int ret = wc_ChaCha20Poly1305_Decrypt(key, iv, inAAD, sizeof(inAAD),
cipher, sizeof(cipher), authTag, plain);

if(ret == MAC_CMP_FAILED_E) {
    // error during authentication
} else if( ret != 0) {
    // error with function arguments
}
```

```
int wc_ChaCha20Poly1305_Encrypt_ex(
    ChaCha * chacha,
    Poly1305 * poly,
    byte * out,
    const byte * in,
    word32 sz,
    const byte * nonce,
    byte * tag,
    const byte * aad,
    word32 aadSz
)
```

This function performs the same AEAD encryption as wc_ChaCha20Poly1305_Encrypt, but takes a caller-owned ChaCha20 context whose key has already been set (with wc_Chacha_SetKey) plus a Poly1305 context, instead of a raw key. This lets the ChaCha20 key be set once and reused across many records, varying only the per-record nonce - it is intended for protocol record layers such as TLS. A fresh per-record Poly1305 key is derived from the ChaCha20 keystream, so this call re-keys the supplied Poly1305 context; the caller does not key it. The 16 byte authentication tag over the AAD and ciphertext is written to tag.

Parameters:

- **chacha** pointer to a ChaCha20 context already keyed with wc_Chacha_SetKey
- **poly** pointer to a Poly1305 context used for the per-record MAC; it is re-keyed internally on each call
- **out** pointer to the buffer in which to store the ciphertext (sz bytes)
- **in** pointer to the buffer containing the plaintext to encrypt
- **sz** the length in bytes of the plaintext to encrypt
- **nonce** pointer to the 12 byte per-record nonce
- **tag** pointer to a 16 byte buffer in which to store the authentication tag
- **aad** pointer to the buffer containing arbitrary length additional authenticated data (AAD)
- **aadSz** length of the input AAD

See:

- wc_ChaCha20Poly1305_Decrypt_ex
- wc_ChaCha20Poly1305_Encrypt
- wc_Chacha_SetKey

Return:

- 0 Returned upon successfully encrypting the message
- BAD_FUNC_ARG Returned if a required pointer argument is NULL (with its matching length nonzero) or otherwise invalid

Example

```
ChaCha chacha;
Poly1305 poly;
byte key[]   = { // initialize 32 byte key };
byte nonce[] = { // initialize 12 byte per-record nonce };
byte aad[]   = { // initialize AAD };
```

```
byte plain[] = { // initialize message to encrypt };
byte cipher[sizeof(plain)];
byte authTag[16];

wc_Chacha_SetKey(&chacha, key, sizeof(key));   // once, then reuse
int ret = wc_ChaCha20Poly1305_Encrypt_ex(&chacha, &poly, cipher, plain,
    sizeof(plain), nonce, authTag, aad, sizeof(aad));
if (ret != 0) {
    // error running encrypt
}
```

```
int wc_ChaCha20Poly1305_Decrypt_ex(
    ChaCha * chacha,
    Poly1305 * poly,
    byte * out,
    const byte * in,
    word32 sz,
    const byte * nonce,
    const byte * tag,
    const byte * aad,
    word32 aadSz
)
```

This function is the decryption counterpart of wc_ChaCha20Poly1305_Encrypt_ex. It takes a caller-owned ChaCha20 context whose key has already been set (with wc_Chacha_SetKey) plus a Poly1305 context, decrypts in to out, and verifies the Poly1305 tag over the AAD and ciphertext. On tag mismatch it returns MAC_CMP_FAILED_E and zeroizes the output buffer, so no unauthenticated plaintext is released. The ChaCha20 key is reused across records, varying only the per-record nonce; the Poly1305 context is re-keyed internally on each call. out may alias in (in-place decryption is supported).

Parameters:

- **chacha** pointer to a ChaCha20 context already keyed with wc_Chacha_SetKey
- **poly** pointer to a Poly1305 context used for the per-record MAC; it is re-keyed internally on each call
- **out** pointer to the buffer in which to store the plaintext (sz bytes)
- **in** pointer to the buffer containing the ciphertext to decrypt
- **sz** the length in bytes of the ciphertext to decrypt
- **nonce** pointer to the 12 byte per-record nonce
- **tag** pointer to the 16 byte authentication tag to verify
- **aad** pointer to the buffer containing arbitrary length additional authenticated data (AAD)
- **aadSz** length of the input AAD

See:

- wc_ChaCha20Poly1305_Encrypt_ex
- wc_ChaCha20Poly1305_Decrypt
- wc_Chacha_SetKey

Return:

- 0 Returned upon successfully decrypting and authenticating the message
- MAC_CMP_FAILED_E Returned if the computed authentication tag does not match the supplied tag; out is zeroized in this case
- BAD_FUNC_ARG Returned if a required pointer argument is NULL (with its matching length nonzero) or otherwise invalid

Example

```
ChaCha chacha;
Poly1305 poly;
byte key[]       = { // initialize 32 byte key };
byte nonce[]     = { // initialize 12 byte per-record nonce };
byte aad[]       = { // initialize AAD };
byte cipher[]    = { // received ciphertext };
byte authTag[16] = { // received authentication tag };
byte plain[sizeof(cipher)];

wc_Chacha_SetKey(&chacha, key, sizeof(key));   // once, then reuse
int ret = wc_ChaCha20Poly1305_Decrypt_ex(&chacha, &poly, plain, cipher,
    sizeof(cipher), nonce, authTag, aad, sizeof(aad));
if (ret == MAC_CMP_FAILED_E) {
    // authentication failed; plain has been zeroized
} else if (ret != 0) {
    // error with function arguments
}
```

```
int wc_ChaCha20Poly1305_CheckTag(
    const byte authTag[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE],
    const byte authTagChk[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE]
)
```

Compares two authentication tags in constant time to prevent timing attacks.

Parameters:

- **authTag** First authentication tag
- **authTagChk** Second authentication tag to compare

See: wc_ChaCha20Poly1305_Decrypt

Return:

- 0 If tags match
- MAC_CMP_FAILED_E If tags do not match

Example

```
byte tag1[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE];
byte tag2[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE];

int ret = wc_ChaCha20Poly1305_CheckTag(tag1, tag2);
```

```
if (ret != 0) {
    // tags do not match
}
```

```
int wc_ChaCha20Poly1305_Init(
    ChaChaPoly_Aead * aead,
    const byte inKey[CHACHA20_POLY1305_AEAD_KEYSIZE],
    const byte inIV[CHACHA20_POLY1305_AEAD_IV_SIZE],
    int isEncrypt
)
```

Initializes a ChaChaPoly_Aead structure for incremental encryption or decryption operations.

Parameters:

- **aead** Pointer to ChaChaPoly_Aead structure to initialize
- **inKey** 32-byte encryption key
- **inIV** 12-byte initialization vector
- **isEncrypt** 1 for encryption, 0 for decryption

See:

- wc_ChaCha20Poly1305_UpdateAad
- wc_ChaCha20Poly1305_UpdateData
- wc_ChaCha20Poly1305_Final

Return:

- 0 On success
- BAD_FUNC_ARG If parameters are invalid

Example

```
ChaChaPoly_Aead aead;
byte key[CHACHA20_POLY1305_AEAD_KEYSIZE];
byte iv[CHACHA20_POLY1305_AEAD_IV_SIZE];

int ret = wc_ChaCha20Poly1305_Init(&aead, key, iv, 1);
if (ret != 0) {
    // error initializing
}
```

```
int wc_ChaCha20Poly1305_UpdateAad(
    ChaChaPoly_Aead * aead,
    const byte * inAAD,
    word32 inAADLen
)
```

Updates the AEAD context with additional authenticated data (AAD). Must be called after Init and before UpdateData.

Parameters:

- **aead** Pointer to initialized ChaChaPoly_Aead structure
- **inAAD** Additional authenticated data
- **inAADLen** Length of AAD in bytes

See:

- wc_ChaCha20Poly1305_Init
- wc_ChaCha20Poly1305_UpdateData

Return:

- 0 On success
- BAD_FUNC_ARG If parameters are invalid

Example

```
ChaChaPoly_Aead aead;
byte aad[]; // AAD data

wc_ChaCha20Poly1305_Init(&aead, key, iv, 1);
int ret = wc_ChaCha20Poly1305_UpdateAad(&aead, aad, sizeof(aad));
if (ret != 0) {
    // error updating AAD
}
```

```
int wc_ChaCha20Poly1305_UpdateData(
    ChaChaPoly_Aead * aead,
    const byte * inData,
    byte * outData,
    word32 dataLen
)
```

Encrypts or decrypts data incrementally. Can be called multiple times to process data in chunks.

Parameters:

- **aead** Pointer to initialized ChaChaPoly_Aead structure
- **inData** Input data (plaintext or ciphertext)
- **outData** Output buffer for result
- **dataLen** Length of data to process

See:

- wc_ChaCha20Poly1305_Init

- wc_ChaCha20Poly1305_Final

Return:

- 0 On success
- BAD_FUNC_ARG If parameters are invalid

Example

```
ChaChaPoly_Aead aead;
byte plain[]; // plaintext
byte cipher[sizeof(plain)];

wc_ChaCha20Poly1305_Init(&aead, key, iv, 1);
wc_ChaCha20Poly1305_UpdateAad(&aead, aad, aadLen);
int ret = wc_ChaCha20Poly1305_UpdateData(&aead, plain,
                                         cipher, sizeof(plain));
```

```
int wc_ChaCha20Poly1305_Final(
    ChaChaPoly_Aead * aead,
    byte outAuthTag[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE]
)
```

Finalizes the AEAD operation and generates the authentication tag.

Parameters:

- **aead** Pointer to ChaChaPoly_Aead structure
- **outAuthTag** Buffer to store 16-byte authentication tag

See:

- wc_ChaCha20Poly1305_Init
- wc_ChaCha20Poly1305_UpdateData

Return:

- 0 On success
- BAD_FUNC_ARG If parameters are invalid

Example

```
ChaChaPoly_Aead aead;
byte authTag[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE];

wc_ChaCha20Poly1305_Init(&aead, key, iv, 1);
wc_ChaCha20Poly1305_UpdateAad(&aead, aad, aadLen);
wc_ChaCha20Poly1305_UpdateData(&aead, plain, cipher, plainLen);
int ret = wc_ChaCha20Poly1305_Final(&aead, authTag);
```

```
int wc_XChaCha20Poly1305_Init(
    ChaChaPoly_Aead * aead,
    const byte * ad,
    word32 ad_len,
    const byte * inKey,
    word32 inKeySz,
    const byte * inIV,
    word32 inIVSz,
    int isEncrypt
)
```

Initializes XChaCha20-Poly1305 AEAD with extended nonce. XChaCha20 uses a 24-byte nonce instead of 12-byte.

Parameters:

- **aead** Pointer to ChaChaPoly_Aead structure
- **ad** Additional authenticated data
- **ad_len** Length of AAD
- **inKey** Encryption key
- **inKeySz** Key size (must be 32)
- **inIV** Initialization vector
- **inIVSz** IV size (must be 24 for XChaCha20)
- **isEncrypt** 1 for encryption, 0 for decryption

See:

- wc_XChaCha20Poly1305_Encrypt
- wc_XChaCha20Poly1305_Decrypt

Return:

- 0 On success
- BAD_FUNC_ARG If parameters are invalid

Example

```
ChaChaPoly_Aead aead;
byte key[32];
byte iv[24];
byte aad[]; // AAD

int ret = wc_XChaCha20Poly1305_Init(&aead, aad, sizeof(aad),
                                    key, 32, iv, 24, 1);
```

```
int wc_XChaCha20Poly1305_Encrypt(
    byte * dst,
    size_t dst_space,
```

```
    const byte * src,
    size_t src_len,
    const byte * ad,
    size_t ad_len,
    const byte * nonce,
    size_t nonce_len,
    const byte * key,
    size_t key_len
)
```

One-shot XChaCha20-Poly1305 encryption with 24-byte nonce.

Parameters:

- **dst** Output buffer for ciphertext and tag
- **dst_space** Size of output buffer
- **src** Input plaintext
- **src_len** Length of plaintext
- **ad** Additional authenticated data
- **ad_len** Length of AAD
- **nonce** 24-byte nonce
- **nonce_len** Nonce length (must be 24)
- **key** 32-byte encryption key
- **key_len** Key length (must be 32)

See: wc_XChaCha20Poly1305_Decrypt

Return:

- 0 On success
- BAD_FUNC_ARG If parameters are invalid
- BUFFER_E If dst_space is insufficient

Example

```
byte key[32], nonce[24];
byte plain[]; // plaintext
byte cipher[sizeof(plain) + 16];

int ret = wc_XChaCha20Poly1305_Encrypt(cipher, sizeof(cipher),
                                       plain, sizeof(plain),
                                       NULL, 0, nonce, 24,
                                       key, 32);
```

```
int wc_XChaCha20Poly1305_Decrypt(
    byte * dst,
    size_t dst_space,
    const byte * src,
    size_t src_len,
    const byte * ad,
    size_t ad_len,
```

```
    const byte * nonce,
    size_t nonce_len,
    const byte * key,
    size_t key_len
)
```

One-shot XChaCha20-Poly1305 decryption with 24-byte nonce.

Parameters:

- **dst** Output buffer for plaintext
- **dst_space** Size of output buffer
- **src** Input ciphertext with tag
- **src_len** Length of ciphertext plus tag
- **ad** Additional authenticated data
- **ad_len** Length of AAD
- **nonce** 24-byte nonce
- **nonce_len** Nonce length (must be 24)
- **key** 32-byte decryption key
- **key_len** Key length (must be 32)

See: wc_XChaCha20Poly1305_Encrypt

Return:

- 0 On success
- BAD_FUNC_ARG If parameters are invalid
- BUFFER_E If dst_space is insufficient
- MAC_CMP_FAILED_E If authentication fails

Example

```
byte key[32], nonce[24];
byte cipher[]; // ciphertext + tag
byte plain[sizeof(cipher) - 16];

int ret = wc_XChaCha20Poly1305_Decrypt(plain, sizeof(plain),
                                       cipher, sizeof(cipher),
                                       NULL, 0, nonce, 24,
                                       key, 32);
if (ret == MAC_CMP_FAILED_E) {
    // authentication failed
}
```

B.18 Algorithm - CMAC

B.17.2.12 function wc_XChaCha20Poly1305_Decrypt

B.18.1 Functions

	Name
int	**wc_InitCmac**(Cmac * cmac, const byte * key, word32 keySz, int type, void * unused)Initialize the Cmac structure with defaults.
int	**wc_InitCmac_ex**(Cmac * cmac, const byte * key, word32 keySz, int type, void * unused, void * heap, int devId)Initialize the Cmac structure with defaults.
int	**wc_CmacUpdate**(Cmac * cmac, const byte * in, word32 inSz)Add Cipher-based Message Authentication Code input data.
int	**wc_CmacFinalNoFree**(Cmac * cmac, byte * out, word32 * outSz)Generate the final result using Cipher-based Message Authentication Code, deferring context cleanup.
int	**wc_CmacFinal**(Cmac * cmac, byte * out, word32 * outSz)Generate the final result using Cipher_based Message Authentication Code, and clean up the context with wc_CmacFree().
int	**wc_CmacFree**(Cmac * cmac)Clean up allocations in a CMAC context.
int	**wc_AesCmacGenerate**(byte * out, word32 * outSz, const byte * in, word32 inSz, const byte * key, word32 keySz)Single shot function for generating a CMAC.
int	**wc_AesCmacVerify**(const byte * check, word32 checkSz, const byte * in, word32 inSz, const byte * key, word32 keySz)Single shot function for validating a CMAC.
int	**wc_CMAC_Grow**(Cmac * cmac, const byte * in, int inSz)Only used with WOLFSSL_HASH_KEEP when hardware requires single-shot and the updates must be cached in memory.
int	**wc_AesCmacGenerate_ex**(Cmac * cmac, byte * out, word32 * outSz, const byte * in, word32 inSz, const byte * key, word32 keySz, void * heap, int devId)Single shot AES-CMAC generation with extended parameters including heap and device ID.
int	**wc_AesCmacVerify_ex**(Cmac * cmac, const byte * check, word32 checkSz, const byte * in, word32 inSz, const byte * key, word32 keySz, void * heap, int devId)Single shot AES-CMAC verification with extended parameters including heap and device ID.

	Name
int	**wc_CmacSetTagLen**(Cmac * cmac, word32 tagLen)This function associates one tag length with the key held by a Cmac object, as SP 800-38B section 5.4 requires. Once set, the final and verify calls for that key accept only that tag length, and this call cannot change or clear it. Starting a new Cmac clears the association. The length is kept on the AES object the Cmac already holds.

B.18.2 Functions Documentation

```
int wc_InitCmac(
    Cmac * cmac,
    const byte * key,
    word32 keySz,
    int type,
    void * unused
)
```

Initialize the Cmac structure with defaults.

Parameters:

- **cmac** pointer to the Cmac structure
- **key** key pointer
- **keySz** size of the key pointer (16, 24 or 32)
- **type** Always WC_CMAC_AES = 1
- **unused** not used, exists for potential future use around compatibility

See:

- wc_InitCmac_ex
- wc_CmacUpdate
- wc_CmacFinal
- wc_CmacFinalNoFree
- wc_CmacFree

Return: 0 on success

Example

```
Cmac cmac[1];
ret = wc_InitCmac(cmac, key, keySz, WC_CMAC_AES, NULL);
if (ret == 0) {
    ret = wc_CmacUpdate(cmac, in, inSz);
}
if (ret == 0) {
    ret = wc_CmacFinal(cmac, out, outSz);
}
```

```
int wc_InitCmac_ex(
    Cmac * cmac,
    const byte * key,
    word32 keySz,
    int type,
    void * unused,
    void * heap,
    int devId
)
```

Initialize the Cmac structure with defaults.

Parameters:

- **cmac** pointer to the Cmac structure
- **key** key pointer
- **keySz** size of the key pointer (16, 24 or 32)
- **type** Always WC_CMAC_AES = 1
- **unused** not used, exists for potential future use around compatibility
- **heap** pointer to the heap hint used for dynamic allocation. Typically used with our static memory option. Can be NULL.
- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID (-2) if not used

See:

- wc_InitCmac_ex
- wc_CmacUpdate
- wc_CmacFinal
- wc_CmacFinalNoFree
- wc_CmacFree

Return: 0 on success

Example

```
Cmac cmac[1];
ret = wc_InitCmac_ex(cmac, key, keySz, WC_CMAC_AES, NULL, NULL, INVALID_DEVID);
if (ret == 0) {
    ret = wc_CmacUpdate(cmac, in, inSz);
}
if (ret == 0) {
    ret = wc_CmacFinal(cmac, out, &outSz);
}
```

```
int wc_CmacUpdate(
    Cmac * cmac,
    const byte * in,
    word32 inSz
)
```

Add Cipher-based Message Authentication Code input data.

Parameters:

- **cmac** pointer to the Cmac structure
- **in** input data to process
- **inSz** size of input data

See:

- wc_InitCmac
- wc_CmacFinal
- wc_CmacFinalNoFree
- wc_CmacFree

Return: 0 on success

Example

```
ret = wc_CmacUpdate(cmac, in, inSz);
```

```
int wc_CmacFinalNoFree(
    Cmac * cmac,
    byte * out,
    word32 * outSz
)
```

Generate the final result using Cipher-based Message Authentication Code, deferring context cleanup.

Parameters:

- **cmac** pointer to the Cmac structure
- **out** pointer to return the result
- **outSz** pointer size of output (in/out)

See:

- wc_InitCmac
- wc_CmacFinal
- wc_CmacFinalNoFree
- wc_CmacFree

Return: 0 on success

Example

```
ret = wc_CmacFinalNoFree(cmac, out, &outSz);
(void)wc_CmacFree(cmac);
```

```
int wc_CmacFinal(
    Cmac * cmac,
    byte * out,
    word32 * outSz
)
```

Generate the final result using Cipher-based Message Authentication Code, and clean up the context with wc_CmacFree().

Parameters:

- **cmac** pointer to the Cmac structure
- **out** pointer to return the result
- **outSz** pointer size of output (in/out)

See:

- wc_InitCmac
- wc_CmacFinalNoFree
- wc_CmacFinalNoFree
- wc_CmacFree

Return: 0 on success

Example

```
ret = wc_CmacFinal(cmac, out, &outSz);
```

```
int wc_CmacFree(
    Cmac * cmac
)
```

Clean up allocations in a CMAC context.

Parameters:

- **cmac** pointer to the Cmac structure

See:

- wc_InitCmac
- wc_CmacFinalNoFree
- wc_CmacFinal
- wc_CmacFree

Return: 0 on success

Example

```
ret = wc_CmacFinalNoFree(cmac, out, &outSz);
(void)wc_CmacFree(cmac);
```

```
int wc_AesCmacGenerate(
    byte * out,
    word32 * outSz,
    const byte * in,
    word32 inSz,
    const byte * key,
    word32 keySz
)
```

Single shot function for generating a CMAC.

Parameters:

- **out** pointer to return the result
- **outSz** pointer size of output (in/out)
- **in** input data to process
- **inSz** size of input data
- **key** key pointer
- **keySz** size of the key pointer (16, 24 or 32)

See: wc_AesCmacVerify

Return: 0 on success

Example

```
ret = wc_AesCmacGenerate(mac, &macSz, msg, msgSz, key, keySz);
```

```
int wc_AesCmacVerify(
    const byte * check,
    word32 checkSz,
    const byte * in,
    word32 inSz,
    const byte * key,
    word32 keySz
)
```

Single shot function for validating a CMAC.

Parameters:

- **check** Expected MAC value to verify
- **checkSz** size of expected MAC value; must be in [WC_CMAC_TAG_MIN_SZ, WC_AES_BLOCK_SIZE]
- **in** input data to process
- **inSz** size of input data
- **key** key pointer
- **keySz** size of the key pointer (16, 24 or 32)

See: wc_AesCmacGenerate

Return:

- 0 on success
- BAD_FUNC_ARG if parameters are invalid
- MAC_CMP_FAILED_E if the supplied tag does not match

Example

```
ret = wc_AesCmacVerify(mac, macSz, msg, msgSz, key, keySz);
```

```
int wc_CMAC_Grow(
    Cmac * cmac,
    const byte * in,
    int inSz
)
```

Only used with WOLFSSL_HASH_KEEP when hardware requires single-shot and the updates must be cached in memory.

Parameters:

- **in** input data to process
- **inSz** size of input data

Return: 0 on success

Example

```
ret = wc_CMAC_Grow(cmac, in, inSz)
```

```
int wc_AesCmacGenerate_ex(
    Cmac * cmac,
    byte * out,
    word32 * outSz,
    const byte * in,
    word32 inSz,
    const byte * key,
    word32 keySz,
    void * heap,
    int devId
)
```

Single shot AES-CMAC generation with extended parameters including heap and device ID.

Parameters:

- **cmac** Pointer to Cmac structure (can be NULL for one-shot)
- **out** Buffer to store MAC output
- **outSz** Pointer to output size (in/out)
- **in** Input data to authenticate
- **inSz** Length of input data
- **key** AES key
- **keySz** Key size (16, 24, or 32 bytes)
- **heap** Heap hint for memory allocation (can be NULL)
- **devId** Device ID for hardware acceleration (use INVALID_DEVID for software)

See:

- wc_AesCmacGenerate
- wc_AesCmacVerify_ex

Return:

- 0 on success
- BAD_FUNC_ARG if parameters are invalid

Example

```
byte mac[AES_BLOCK_SIZE];
word32 macSz = sizeof(mac);
byte key[16], msg[64];

int ret = wc_AesCmacGenerate_ex(NULL, mac, &macSz, msg,
                                sizeof(msg), key, sizeof(key),
                                NULL, INVALID_DEVID);
```

```
int wc_AesCmacVerify_ex(
    Cmac * cmac,
    const byte * check,
    word32 checkSz,
    const byte * in,
    word32 inSz,
    const byte * key,
    word32 keySz,
    void * heap,
    int devId
)
```

Single shot AES-CMAC verification with extended parameters including heap and device ID.

Parameters:

- **cmac** Pointer to Cmac structure
- **check** Expected MAC value to verify
- **checkSz** Size of expected MAC; must be in [WC_CMAC_TAG_MIN_SZ, WC_AES_BLOCK_SIZE]
- **in** Input data to authenticate

- **inSz** Length of input data
- **key** AES key
- **keySz** Key size (16, 24, or 32 bytes)
- **heap** Heap hint for memory allocation (can be NULL)
- **devId** Device ID for hardware acceleration (use INVALID_DEVID for software)

See:

- wc_InitCmac_ex
- wc_AesCmacVerify
- wc_AesCmacGenerate_ex

Return:

- 0 on success
- BAD_FUNC_ARG if parameters are invalid
- MAC_CMP_FAILED_E if MAC verification fails

Example

```
Cmac cmac;
byte mac[WC_AES_BLOCK_SIZE];
byte key[16], msg[64];
int ret;

ret = wc_InitCmac_ex(&cmac, key, sizeof(key), WC_CMAC_AES, NULL,
                     NULL, INVALID_DEVID);
if (ret == 0) {
    ret = wc_AesCmacVerify_ex(&cmac, mac, sizeof(mac), msg,
                              sizeof(msg), key, sizeof(key),
                              NULL, INVALID_DEVID);
}
if (ret == MAC_CMP_FAILED_E) {
    // MAC verification failed
}
```

```
int wc_CmacSetTagLen(
    Cmac * cmac,
    word32 tagLen
)
```

This function associates one tag length with the key held by a Cmac object, as SP 800-38B section 5.4 requires. Once set, the final and verify calls for that key accept only that tag length, and this call cannot change or clear it. Starting a new Cmac clears the association. The length is kept on the AES object the Cmac already holds.

Parameters:

- **cmac** pointer to the Cmac object holding the key
- **tagLen** tag length to associate with the key

See:

- wc_InitCmac
- wc_CmacFinal
- wc_AesSetTagLen

Return:

- 0 On success.
- BAD_FUNC_ARG Returned if cmac is NULL, the length is outside WC_CMAC_TAG_MIN_SZ to WC_CMAC_TAG_MAX_SZ, or a different length is already associated.

Example

```
Cmac cmac;
byte key[] = { some 16, 24, 32 byte key };
if (wc_InitCmac(&cmac, key, sizeof(key), WC_CMAC_AES, NULL) != 0) {
    // failed to set up cmac
}
if (wc_CmacSetTagLen(&cmac, 16) != 0) {
    // failed to associate the tag length
}
```

B.19 Callbacks - CryptoCb

B.18.2.12 function wc_CmacSetTagLen

B.19.1 Functions

	Name
int	**wc_BufferKeyDecrypt**(struct EncryptedInfo * info, byte * der, word32 derSz, const byte * password, int passwordSz, int hashType)This function decrypts an encrypted key buffer using the provided password. It supports encryption algorithms DES, 3DES, and AES. The encryption information is provided in the EncryptedInfo structure.
int	**wc_BufferKeyEncrypt**(struct EncryptedInfo * info, byte * der, word32 derSz, const byte * password, int passwordSz, int hashType)This function encrypts a key buffer using the provided password. It supports encryption algorithms DES, 3DES, and AES. The encryption information is provided in the EncryptedInfo structure.

B.19.2 Functions Documentation

```
int wc_BufferKeyDecrypt(
    struct EncryptedInfo * info,
    byte * der,
    word32 derSz,
    const byte * password,
    int passwordSz,
    int hashType
)
```

This function decrypts an encrypted key buffer using the provided password. It supports encryption algorithms DES, 3DES, and AES. The encryption information is provided in the EncryptedInfo structure.

Parameters:

- **info** pointer to EncryptedInfo structure containing encryption algorithm and parameters
- **der** pointer to the encrypted key buffer
- **derSz** size of the encrypted key buffer
- **password** pointer to the password buffer
- **passwordSz** size of the password
- **hashType** hash algorithm to use for key derivation

See: wc_BufferKeyEncrypt

Return:

- 0 on success
- BAD_FUNC_ARG if der is NULL
- BAD_FUNC_ARG if password is NULL
- BAD_FUNC_ARG if info is NULL
- BAD_FUNC_ARG if info->keySz is 0
- BAD_FUNC_ARG if info->keySz > WC_MAX_SYM_KEY_SIZE
- ALGO_ID_E if info->cipherType is not one of the cipher types this function handles
- BUFFER_E if IV buffer is too small
- NOT_COMPILED_IN if info->cipherType is a cipher type this function handles, but support for it was disabled in this build
- Negative value on error

Example

```
EncryptedInfo info;
byte encryptedKey[32]; // encrypted key data, decrypted in place
byte password[] = "mypassword";

XMEMSET(&info, 0, sizeof(info));
info.cipherType = WC_CIPHER_AES_CBC;
info.keySz = 16;       // size of the key derived from the password

// info.iv holds the IV as hex characters, the way it appears in a PEM
// DEK-Info header.  It is Base16-decoded in place, and must decode to
// at least PKCS5_SALT_SZ bytes.  The decoded bytes are used both as the
// key derivation salt and as the cipher IV.
XMEMCPY(info.iv, "0123456789ABCDEF0123456789ABCDEF", 32);
```

```
info.ivSz = 32;        // length of the hex string, not the decoded IV

int ret = wc_BufferKeyDecrypt(&info, encryptedKey,
                              sizeof(encryptedKey), password,
                              sizeof(password)-1, WC_SHA256);
if (ret != 0) {
    // decryption error
}
```

```
int wc_BufferKeyEncrypt(
    struct EncryptedInfo * info,
    byte * der,
    word32 derSz,
    const byte * password,
    int passwordSz,
    int hashType
)
```

This function encrypts a key buffer using the provided password. It supports encryption algorithms DES, 3DES, and AES. The encryption information is provided in the EncryptedInfo structure.

Parameters:

- **info** pointer to EncryptedInfo structure containing encryption algorithm and parameters
- **der** pointer to the key buffer to encrypt
- **derSz** size of the key buffer
- **password** pointer to the password buffer
- **passwordSz** size of the password
- **hashType** hash algorithm to use for key derivation

See: wc_BufferKeyDecrypt

Return:

- 0 on success
- BAD_FUNC_ARG if der is NULL
- BAD_FUNC_ARG if password is NULL
- BAD_FUNC_ARG if info is NULL
- BAD_FUNC_ARG if info->keySz is 0
- BAD_FUNC_ARG if info->keySz > WC_MAX_SYM_KEY_SIZE
- BAD_FUNC_ARG if info->ivSz < PKCS5_SALT_SZ
- ALGO_ID_E if info->cipherType is not one of the cipher types this function handles
- NOT_COMPILED_IN if info->cipherType is a cipher type this function handles, but support for it was disabled in this build
- Negative value on error

Example

```
EncryptedInfo info;
byte key[32]; // key data to encrypt, encrypted in place
byte password[] = "mypassword";
```

```
XMEMSET(&info, 0, sizeof(info));
info.cipherType = WC_CIPHER_AES_CBC;
info.keySz = 16;   // size of the key derived from the password
info.ivSz  = 16;   // must be at least PKCS5_SALT_SZ

// Unlike wc_BufferKeyDecrypt, info.iv holds raw bytes here.  Fill it
// with info.ivSz random bytes; the first PKCS5_SALT_SZ of them are used
// as the key derivation salt, and the buffer is also used as the
// cipher IV.
WC_RNG rng;
wc_InitRng(&rng);
wc_RNG_GenerateBlock(&rng, info.iv, info.ivSz);

int ret = wc_BufferKeyEncrypt(&info, key, sizeof(key), password,
                              sizeof(password)-1, WC_SHA256);
if (ret != 0) {
    // encryption error
}
wc_FreeRng(&rng);
```

B.20 Algorithms - Curve25519

B.19.2.2 function wc_BufferKeyEncrypt

B.20.1 Functions

	Name
int	**wc_curve25519_make_key**(WC_RNG * rng, int keysize, curve25519_key * key)This function generates a Curve25519 key using the given random number generator, rng, of the size given (keysize), and stores it in the given curve25519_key structure. It should be called after the key structure has been initialized through wc_curve25519_init().
int	**wc_curve25519_shared_secret**(curve25519_key * private_key, curve25519_key * public_key, byte * out, word32 * outlen)This function computes a shared secret key given a secret private key and a received public key. It stores the generated secret key in the buffer out and assigns the length of the secret key to outlen. Only supports big endian.

	Name
int	**wc_curve25519_shared_secret_ex**(curve25519_key * private_key, curve25519_key * public_key, byte * out, word32 * outlen, int endian)This function computes a shared secret key given a secret private key and a received public key. It stores the generated secret key in the buffer out and assigns the length of the secret key to outlen. Supports both big and little endian.
int	**wc_curve25519_init**(curve25519_key * key)This function initializes a Curve25519 key. It should be called before generating a key for the structure.
void	**wc_curve25519_free**(curve25519_key * key)This function frees a Curve25519 object.
int	**wc_curve25519_import_private**(const byte * priv, word32 privSz, curve25519_key * key)This function imports a curve25519 private key only. (Big endian).
int	**wc_curve25519_import_private_ex**(const byte * priv, word32 privSz, curve25519_key * key, int endian)curve25519 private key import only. (Big or Little endian).
int	**wc_curve25519_import_private_raw**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, curve25519_key * key)This function imports a public-private key pair into a curve25519_key structure. Big endian only.
int	**wc_curve25519_import_private_raw_ex**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, curve25519_key * key, int endian)This function imports a public-private key pair into a curve25519_key structure. Supports both big and little endian.
int	**wc_curve25519_export_private_raw**(curve25519_key * key, byte * out, word32 * outLen)This function exports a private key from a curve25519_key structure and stores it in the given out buffer. It also sets outLen to be the size of the exported key. Big Endian only.
int	**wc_curve25519_export_private_raw_ex**(curve25519_key * key, byte * out, word32 * outLen, int endian)This function exports a private key from a curve25519_key structure and stores it in the given out buffer. It also sets outLen to be the size of the exported key. Can specify whether it's big or little endian.
int	**wc_curve25519_import_public**(const byte * in, word32 inLen, curve25519_key * key)This function imports a public key from the given in buffer and stores it in the curve25519_key structure.

	Name
int	**wc_curve25519_import_public_ex**(const byte * in, word32 inLen, curve25519_key * key, int endian)This function imports a public key from the given in buffer and stores it in the curve25519_key structure.
int	**wc_curve25519_check_public**(const byte * pub, word32 pubSz, int endian)This function checks that a public key buffer holds a valid Curve25519 key value given the endian ordering.
int	**wc_curve25519_export_public**(curve25519_key * key, byte * out, word32 * outLen)This function exports a public key from the given key structure and stores the result in the out buffer. Big endian only.
int	**wc_curve25519_export_public_ex**(curve25519_key * key, byte * out, word32 * outLen, int endian)This function exports a public key from the given key structure and stores the result in the out buffer. Supports both big and little endian.
int	**wc_curve25519_export_key_raw**(curve25519_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz)Export Curve25519 key pair. Big endian only.
int	**wc_curve25519_export_key_raw_ex**(curve25519_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz, int endian)Export curve25519 key pair. Big or little endian.
int	**wc_curve25519_size**(curve25519_key * key)This function returns the key size of the given key structure.
int	**wc_curve25519_make_pub**(int public_size, byte * pub, int private_size, const byte * priv)This function generates a Curve25519 public key from a given private key. This is a lower-level function that operates directly on byte buffers rather than curve25519_key structures.
int	**wc_curve25519_make_pub_blind**(int public_size, byte * pub, int private_size, const byte * priv, WC_RNG * rng)This function generates a Curve25519 public key from a given private key with blinding to resist side-channel attacks. This adds randomization to the scalar multiplication operation.

	Name
int	**wc_curve25519_generic**(int public_size, byte * pub, int private_size, const byte * priv, int basepoint_size, const byte * basepoint)This function performs a generic Curve25519 scalar multiplication with a custom basepoint. This allows computing scalar * basepoint for any basepoint, not just the standard generator.
int	**wc_curve25519_generic_blind**(int public_size, byte * pub, int private_size, const byte * priv, int basepoint_size, const byte * basepoint, WC_RNG * rng)This function performs a generic Curve25519 scalar multiplication with a custom basepoint and blinding to resist side-channel attacks.
int	**wc_curve25519_make_priv**(WC_RNG * rng, int keysize, byte * priv)This function generates a Curve25519 private key using the given random number generator. This is a lower-level function that generates only the private key bytes.
int	**wc_curve25519_init_ex**(curve25519_key * key, void * heap, int devId)This function initializes a Curve25519 key with extended parameters, allowing specification of custom heap and device ID for hardware acceleration.
int	**wc_curve25519_set_rng**(curve25519_key * key, WC_RNG * rng)This function sets the RNG to be used with a Curve25519 key. This is useful for operations that require randomness such as blinded scalar multiplication.
curve25519_key *	**wc_curve25519_new**(void * heap, int devId, int * result_code)This function allocates and initializes a new Curve25519 key structure with extended parameters. The caller is responsible for freeing the key with wc_curve25519_delete. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.
int	**wc_curve25519_delete**(curve25519_key * key, curve25519_key ** key_p)This function frees a Curve25519 key structure that was allocated with wc_curve25519_new and sets the pointer to NULL. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.

B.20.2 Functions Documentation

```
int wc_curve25519_make_key(
    WC_RNG * rng,
```

```
    int keysize,
    curve25519_key * key
)
```

This function generates a Curve25519 key using the given random number generator, rng, of the size given (keysize), and stores it in the given curve25519_key structure. It should be called after the key structure has been initialized through wc_curve25519_init().

Parameters:

- **rng** Pointer to the RNG object used to generate the ecc key.
- **keysize** Size of the key to generate. Must be 32 bytes for curve25519.
- **key** Pointer to the curve25519_key structure in which to store the generated key.

See: wc_curve25519_init

Return:

- 0 Returned on successfully generating the key and and storing it in the given curve25519_key structure.
- ECC_BAD_ARG_E Returned if the input keysize does not correspond to the keysize for a curve25519 key (32 bytes).
- RNG_FAILURE_E Returned if the rng internal status is not DRBG_OK or if there is in generating the next random block with rng.
- BAD_FUNC_ARG Returned if any of the input parameters passed in are NULL.

Example

```
int ret;

curve25519_key key;
wc_curve25519_init(&key); // initialize key
WC_RNG rng;
wc_InitRng(&rng); // initialize random number generator

ret = wc_curve25519_make_key(&rng, 32, &key);
if (ret != 0) {
    // error making Curve25519 key
}
```

```
int wc_curve25519_shared_secret(
    curve25519_key * private_key,
    curve25519_key * public_key,
    byte * out,
    word32 * outlen
)
```

This function computes a shared secret key given a secret private key and a received public key. It stores the generated secret key in the buffer out and assigns the length of the secret key to outlen. Only supports big endian.

Parameters:

- **private_key** Pointer to the curve25519_key structure initialized with the user's private key.
- **public_key** Pointer to the curve25519_key structure containing the received public key.
- **out** Pointer to a buffer in which to store the 32 byte computed secret key.
- **outlen** Pointer in which to store the length written to the output buffer.

See:

- wc_curve25519_init
- wc_curve25519_make_key
- wc_curve25519_shared_secret_ex

Return:

- 0 Returned on successfully computing a shared secret key.
- BAD_FUNC_ARG Returned if any of the input parameters passed in are NULL.
- ECC_BAD_ARG_E Returned if the first bit of the public key is set, to avoid implementation fingerprinting.

Example

```
int ret;

byte sharedKey[32];
word32 keySz;
curve25519_key privKey, pubKey;
// initialize both keys

ret = wc_curve25519_shared_secret(&privKey, &pubKey, sharedKey, &keySz);
if (ret != 0) {
    // error generating shared key
}
```

```
int wc_curve25519_shared_secret_ex(
    curve25519_key * private_key,
    curve25519_key * public_key,
    byte * out,
    word32 * outlen,
    int endian
)
```

This function computes a shared secret key given a secret private key and a received public key. It stores the generated secret key in the buffer out and assigns the length of the secret key to outlen. Supports both big and little endian.

Parameters:

- **private_key** Pointer to the curve25519_key structure initialized with the user's private key.
- **public_key** Pointer to the curve25519_key structure containing the received public key.
- **out** Pointer to a buffer in which to store the 32 byte computed secret key.

- **outlen** Pointer in which to store the length written to the output buffer.
- **endian** EC25519_BIG_ENDIAN or EC25519_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve25519_init
- wc_curve25519_make_key
- wc_curve25519_shared_secret

Return:

- 0 Returned on successfully computing a shared secret key.
- BAD_FUNC_ARG Returned if any of the input parameters passed in are NULL.
- ECC_BAD_ARG_E Returned if the first bit of the public key is set, to avoid implementation fingerprinting.

Example

```
int ret;

byte sharedKey[32];
word32 keySz;

curve25519_key privKey, pubKey;
// initialize both keys

ret = wc_curve25519_shared_secret_ex(&privKey, &pubKey, sharedKey, &keySz,
         EC25519_BIG_ENDIAN);
if (ret != 0) {
    // error generating shared key
}
```

```
int wc_curve25519_init(
    curve25519_key * key
)
```

This function initializes a Curve25519 key. It should be called before generating a key for the structure.

Parameters:

- **key** Pointer to the curve25519_key structure to initialize.

See: wc_curve25519_make_key

Return:

- 0 Returned on successfully initializing the curve25519_key structure.
- BAD_FUNC_ARG Returned when key is NULL.

Example

```
curve25519_key key;
wc_curve25519_init(&key); // initialize key
// make key and proceed to encryption
```

```
void wc_curve25519_free(
    curve25519_key * key
)
```

This function frees a Curve25519 object.

Parameters:

- **key** Pointer to the key object to free.

See:

- wc_curve25519_init
- wc_curve25519_make_key

Example

```
curve25519_key privKey;
// initialize key, use it to generate shared secret key
wc_curve25519_free(&privKey);
```

```
int wc_curve25519_import_private(
    const byte * priv,
    word32 privSz,
    curve25519_key * key
)
```

This function imports a curve25519 private key only. (Big endian).

Parameters:

- **priv** Pointer to a buffer containing the private key to import.
- **privSz** Length of the private key to import.
- **key** Pointer to the structure in which to store the imported key.

See:

- wc_curve25519_import_private_ex
- wc_curve25519_size

Return:

- 0 Returned on successfully importing private key.
- BAD_FUNC_ARG Returns if key or priv is null.
- ECC_BAD_ARG_E Returns if privSz is not equal to CURVE25519_KEY_SIZE.

Example

```
int ret;

byte priv[] = { Contents of private key };
curve25519_key key;
wc_curve25519_init(&key);

ret = wc_curve25519_import_private(priv, sizeof(priv), &key);
if (ret != 0) {
    // error importing keys
}
```

```
int wc_curve25519_import_private_ex(
    const byte * priv,
    word32 privSz,
    curve25519_key * key,
    int endian
)
```

curve25519 private key import only. (Big or Little endian).

Parameters:

- **priv** Pointer to a buffer containing the private key to import.
- **privSz** Length of the private key to import.
- **key** Pointer to the structure in which to store the imported key.
- **endian** EC25519_BIG_ENDIAN or EC25519_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve25519_import_private
- wc_curve25519_size

Return:

- 0 Returned on successfully importing private key.
- BAD_FUNC_ARG Returns if key or priv is null.
- ECC_BAD_ARG_E Returns if privSz is not equal to CURVE25519_KEY_SIZE.

Example

```
int ret;

byte priv[] = { // Contents of private key };
curve25519_key key;
wc_curve25519_init(&key);

ret = wc_curve25519_import_private_ex(priv, sizeof(priv), &key,
        EC25519_BIG_ENDIAN);
if (ret != 0) {
    // error importing keys
}
```

```
int wc_curve25519_import_private_raw(
    const byte * priv,
    word32 privSz,
    const byte * pub,
    word32 pubSz,
    curve25519_key * key
)
```

This function imports a public-private key pair into a curve25519_key structure. Big endian only.

Parameters:

- **priv** Pointer to a buffer containing the private key to import.
- **privSz** Length of the private key to import.
- **pub** Pointer to a buffer containing the public key to import.
- **pubSz** Length of the public key to import.
- **key** Pointer to the structure in which to store the imported keys.

See:

- wc_curve25519_init
- wc_curve25519_make_key
- wc_curve25519_import_public
- wc_curve25519_export_private_raw

Return:

- 0 Returned on importing into the curve25519_key structure
- BAD_FUNC_ARG Returns if any of the input parameters are null.
- ECC_BAD_ARG_E Returned if the input key's key size does not match the public or private key sizes.

Example

```
int ret;

byte priv[32];
byte pub[32];
```

```
// initialize with public and private keys
curve25519_key key;

wc_curve25519_init(&key);
// initialize key

ret = wc_curve25519_import_private_raw(&priv, sizeof(priv), pub,
        sizeof(pub), &key);
if (ret != 0) {
    // error importing keys
}
```

```
int wc_curve25519_import_private_raw_ex(
    const byte * priv,
    word32 privSz,
    const byte * pub,
    word32 pubSz,
    curve25519_key * key,
    int endian
)
```

This function imports a public-private key pair into a curve25519_key structure. Supports both big and little endian.

Parameters:

- **priv** Pointer to a buffer containing the private key to import.
- **privSz** Length of the private key to import.
- **pub** Pointer to a buffer containing the public key to import.
- **pubSz** Length of the public key to import.
- **key** Pointer to the structure in which to store the imported keys.
- **endian** EC25519_BIG_ENDIAN or EC25519_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve25519_init
- wc_curve25519_make_key
- wc_curve25519_import_public
- wc_curve25519_export_private_raw
- wc_curve25519_import_private_raw

Return:

- 0 Returned on importing into the curve25519_key structure
- BAD_FUNC_ARG Returns if any of the input parameters are null.
- ECC_BAD_ARG_E Returned if or the input key's key size does not match the public or private key sizes

Example

```
int ret;
byte priv[32];
byte pub[32];
// initialize with public and private keys
curve25519_key key;

wc_curve25519_init(&key);
// initialize key

ret = wc_curve25519_import_private_raw_ex(&priv, sizeof(priv), pub,
        sizeof(pub), &key, EC25519_BIG_ENDIAN);
if (ret != 0) {
    // error importing keys
}
```

```
int wc_curve25519_export_private_raw(
    curve25519_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports a private key from a curve25519_key structure and stores it in the given out buffer. It also sets outLen to be the size of the exported key. Big Endian only.

Parameters:

- **key** Pointer to the structure from which to export the key.
- **out** Pointer to the buffer in which to store the exported key.
- **outLen** On in, is the size of the out in bytes. On out, will store the bytes written to the output buffer.

See:

- wc_curve25519_init
- wc_curve25519_make_key
- wc_curve25519_import_private_raw
- wc_curve25519_export_private_raw_ex

Return:

- 0 Returned on successfully exporting the private key from the curve25519_key structure.
- BAD_FUNC_ARG Returned if any input parameters are NULL.
- ECC_BAD_ARG_E Returned if *outLen is less than wc_curve25519_size().

Example

```
int ret;
byte priv[32];
word32 privSz;
```

```
curve25519_key key;
// initialize and make key

ret = wc_curve25519_export_private_raw(&key, priv, &privSz);
if (ret != 0) {
    // error exporting key
}
```

```
int wc_curve25519_export_private_raw_ex(
    curve25519_key * key,
    byte * out,
    word32 * outLen,
    int endian
)
```

This function exports a private key from a curve25519_key structure and stores it in the given out buffer. It also sets outLen to be the size of the exported key. Can specify whether it's big or little endian.

Parameters:

- **key** Pointer to the structure from which to export the key.
- **out** Pointer to the buffer in which to store the exported key.
- **outLen** On in, is the size of the out in bytes. On out, will store the bytes written to the output buffer.
- **endian** EC25519_BIG_ENDIAN or EC25519_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve25519_init
- wc_curve25519_make_key
- wc_curve25519_import_private_raw
- wc_curve25519_export_private_raw
- wc_curve25519_size

Return:

- 0 Returned on successfully exporting the private key from the curve25519_key structure.
- BAD_FUNC_ARG Returned if any input parameters are NULL.
- ECC_BAD_ARG_E Returned if *outLen is less than wc_curve25519_size().

Example

```
int ret;

byte priv[32];
word32 privSz;
curve25519_key key;
// initialize and make key
ret = wc_curve25519_export_private_raw_ex(&key, priv, &privSz,
        EC25519_BIG_ENDIAN);
```

```
if (ret != 0) {
    // error exporting key
}
```

```
int wc_curve25519_import_public(
    const byte * in,
    word32 inLen,
    curve25519_key * key
)
```

This function imports a public key from the given in buffer and stores it in the curve25519_key structure.

Parameters:

- **in** Pointer to the buffer containing the public key to import.
- **inLen** Length of the public key to import.
- **key** Pointer to the curve25519_key structure in which to store the key.

See:

- wc_curve25519_init
- wc_curve25519_export_public
- wc_curve25519_import_private_raw
- wc_curve25519_import_public_ex
- wc_curve25519_check_public
- wc_curve25519_size

Return:

- 0 Returned on successfully importing the public key into the curve25519_key structure.
- ECC_BAD_ARG_E Returned if the inLen parameter does not match the key size of the key structure.
- BAD_FUNC_ARG Returned if any of the input parameters are NULL.

Example

```
int ret;

byte pub[32];
// initialize pub with public key

curve25519_key key;
// initialize key

ret = wc_curve25519_import_public(pub,sizeof(pub), &key);
if (ret != 0) {
    // error importing key
}
```

```
int wc_curve25519_import_public_ex(
    const byte * in,
    word32 inLen,
    curve25519_key * key,
    int endian
)
```

This function imports a public key from the given in buffer and stores it in the curve25519_key structure.

Parameters:

- **in** Pointer to the buffer containing the public key to import.
- **inLen** Length of the public key to import.
- **key** Pointer to the curve25519_key structure in which to store the key.
- **endian** EC25519_BIG_ENDIAN or EC25519_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve25519_init
- wc_curve25519_export_public
- wc_curve25519_import_private_raw
- wc_curve25519_import_public
- wc_curve25519_check_public
- wc_curve25519_size

Return:

- 0 Returned on successfully importing the public key into the curve25519_key structure.
- ECC_BAD_ARG_E Returned if the inLen parameter does not match the key size of the key structure.
- BAD_FUNC_ARG Returned if any of the input parameters are NULL.

Example

```
int ret;

byte pub[32];
// initialize pub with public key
curve25519_key key;
// initialize key

ret = wc_curve25519_import_public_ex(pub, sizeof(pub), &key,
        EC25519_BIG_ENDIAN);
if (ret != 0) {
    // error importing key
}
```

```
int wc_curve25519_check_public(
    const byte * pub,
```

```
    word32 pubSz,
    int endian
)
```

This function checks that a public key buffer holds a valid Curve25519 key value given the endian ordering.

Parameters:

- **pub** Pointer to the buffer containing the public key to check.
- **pubSz** Length of the public key to check.
- **endian** EC25519_BIG_ENDIAN or EC25519_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve25519_init
- wc_curve25519_import_public
- wc_curve25519_import_public_ex
- wc_curve25519_size

Return:

- 0 Returned when the public key value is valid.
- ECC_BAD_ARG_E Returned if the public key value is not valid.
- BAD_FUNC_ARG Returned if any of the input parameters are NULL.

Example

```
int ret;

byte pub[] = { Contents of public key };

ret = wc_curve25519_check_public_ex(pub, sizeof(pub), EC25519_BIG_ENDIAN);
if (ret != 0) {
    // error importing key
}
```

```
int wc_curve25519_export_public(
    curve25519_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports a public key from the given key structure and stores the result in the out buffer. Big endian only.

Parameters:

- **key** Pointer to the curve25519_key structure in from which to export the key.
- **out** Pointer to the buffer in which to store the public key.

- **outLen** On in, is the size of the out in bytes. On out, will store the bytes written to the output buffer.

See:

- wc_curve25519_init
- wc_curve25519_export_private_raw
- wc_curve25519_import_public

Return:

- 0 Returned on successfully exporting the public key from the curve25519_key structure.
- ECC_BAD_ARG_E Returned if outLen is less than CURVE25519_PUB_KEY_SIZE.
- BAD_FUNC_ARG Returned if any of the input parameters are NULL.

Example

```
int ret;

byte pub[32];
int pubSz;

curve25519_key key;
// initialize and make key
ret = wc_curve25519_export_public(&key, pub, &pubSz);
if (ret != 0) {
    // error exporting key
}
```

```
int wc_curve25519_export_public_ex(
    curve25519_key * key,
    byte * out,
    word32 * outLen,
    int endian
)
```

This function exports a public key from the given key structure and stores the result in the out buffer. Supports both big and little endian.

Parameters:

- **key** Pointer to the curve25519_key structure in from which to export the key.
- **out** Pointer to the buffer in which to store the public key.
- **outLen** On in, is the size of the out in bytes. On out, will store the bytes written to the output buffer.
- **endian** EC25519_BIG_ENDIAN or EC25519_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve25519_init

- wc_curve25519_export_private_raw
- wc_curve25519_import_public

Return:

- 0 Returned on successfully exporting the public key from the curve25519_key structure.
- ECC_BAD_ARG_E Returned if outLen is less than CURVE25519_PUB_KEY_SIZE.
- BAD_FUNC_ARG Returned if any of the input parameters are NULL.
- NO_VALID_DEVID Returned when the public point still has to be derived, the build is WOLF_CRYPTO_CB_ONLY_CURVE25519 and the key is not bound to a device. Such a key has no software path to fall back on and is never offloaded to an arbitrary registered device, so it must be created with wc_curve25519_init_ex() and a real devId.

Example

```
int ret;

byte pub[32];
int pubSz;

curve25519_key key;
// initialize and make key

ret = wc_curve25519_export_public_ex(&key, pub, &pubSz, EC25519_BIG_ENDIAN);
if (ret != 0) {
    // error exporting key
}
```

```
int wc_curve25519_export_key_raw(
    curve25519_key * key,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz
)
```

Export Curve25519 key pair. Big endian only.

Parameters:

- **key** Pointer to the curve25519_key structure in from which to export the key pair.
- **priv** Pointer to the buffer in which to store the private key.
- **privSz** On in, is the size of the priv buffer in bytes. On out, will store the bytes written to the priv buffer.
- **pub** Pointer to the buffer in which to store the public key.
- **pubSz** On in, is the size of the pub buffer in bytes. On out, will store the bytes written to the pub buffer.

See:

- wc_curve25519_export_key_raw_ex
- wc_curve25519_export_private_raw

Return:

- 0 Returned on successfully exporting the key pair from the curve25519_key structure.
- BAD_FUNC_ARG Returned if any input parameters are NULL.
- ECC_BAD_ARG_E Returned if privSz is less than CURVE25519_KEY_SIZE or pubSz is less than CURVE25519_PUB_KEY_SIZE.

Example

```
int ret;

byte pub[32];
byte priv[32];
int pubSz;
int privSz;

curve25519_key key;
// initialize and make key

ret = wc_curve25519_export_key_raw(&key, priv, &privSz, pub, &pubSz);
if (ret != 0) {
    // error exporting key
}
```

```
int wc_curve25519_export_key_raw_ex(
    curve25519_key * key,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz,
    int endian
)
```

Export curve25519 key pair. Big or little endian.

Parameters:

- **key** Pointer to the curve25519_key structure in from which to export the key pair.
- **priv** Pointer to the buffer in which to store the private key.
- **privSz** On in, is the size of the priv buffer in bytes. On out, will store the bytes written to the priv buffer.
- **pub** Pointer to the buffer in which to store the public key.
- **pubSz** On in, is the size of the pub buffer in bytes. On out, will store the bytes written to the pub buffer.
- **endian** EC25519_BIG_ENDIAN or EC25519_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve25519_export_key_raw
- wc_curve25519_export_private_raw_ex
- wc_curve25519_export_public_ex

Return:

- 0 Returned on successfully exporting the key pair from the curve25519_key structure.
- BAD_FUNC_ARG Returned if any input parameters are NULL.
- ECC_BAD_ARG_E Returned if privSz is less than CURVE25519_KEY_SIZE or pubSz is less than CURVE25519_PUB_KEY_SIZE.

Example

```
int ret;

byte pub[32];
byte priv[32];
int pubSz;
int privSz;

curve25519_key key;
// initialize and make key

ret = wc_curve25519_export_key_raw_ex(&key, priv, &privSz, pub, &pubSz,
        EC25519_BIG_ENDIAN);
if (ret != 0) {
    // error exporting key
}
```

```
int wc_curve25519_size(
    curve25519_key * key
)
```

This function returns the key size of the given key structure.

Parameters:

- **key** Pointer to the curve25519_key structure in for which to determine the key size.

See:

- wc_curve25519_init
- wc_curve25519_make_key

Return:

- Success Given a valid, initialized curve25519_key structure, returns the size of the key.
- 0 Returned if key is NULL

Example

```
int keySz;

curve25519_key key;
// initialize and make key

keySz = wc_curve25519_size(&key);
```

```
int wc_curve25519_make_pub(
    int public_size,
    byte * pub,
    int private_size,
    const byte * priv
)
```

This function generates a Curve25519 public key from a given private key. This is a lower-level function that operates directly on byte buffers rather than curve25519_key structures.

Parameters:

- **public_size** Size of the public key buffer (must be 32)
- **pub** Pointer to buffer to store the public key
- **private_size** Size of the private key (must be 32)
- **priv** Pointer to buffer containing the private key

See:

- wc_curve25519_make_key
- wc_curve25519_make_pub_blind

Return:

- 0 On successfully generating the public key
- ECC_BAD_ARG_E If the key sizes are invalid
- BAD_FUNC_ARG If any input parameters are NULL

Example

```
byte priv[CURVE25519_KEYSIZE];
byte pub[CURVE25519_KEYSIZE];

// initialize priv with private key
int ret = wc_curve25519_make_pub(sizeof(pub), pub, sizeof(priv),
                                 priv);
if (ret != 0) {
    // error generating public key
}
```

```
int wc_curve25519_make_pub_blind(
    int public_size,
    byte * pub,
    int private_size,
    const byte * priv,
    WC_RNG * rng
)
```

This function generates a Curve25519 public key from a given private key with blinding to resist side-channel attacks. This adds randomization to the scalar multiplication operation.

Parameters:

- **public_size** Size of the public key buffer (must be 32)
- **pub** Pointer to buffer to store the public key
- **private_size** Size of the private key (must be 32)
- **priv** Pointer to buffer containing the private key
- **rng** Pointer to initialized RNG for blinding

See:

- wc_curve25519_make_pub
- wc_curve25519_generic_blind

Return:

- 0 On successfully generating the public key
- ECC_BAD_ARG_E If the key sizes are invalid
- BAD_FUNC_ARG If any input parameters are NULL

Example

```
WC_RNG rng;
byte priv[CURVE25519_KEYSIZE];
byte pub[CURVE25519_KEYSIZE];

wc_InitRng(&rng);
// initialize priv with private key
int ret = wc_curve25519_make_pub_blind(sizeof(pub), pub,
                                       sizeof(priv), priv, &rng);
if (ret != 0) {
    // error generating public key
}
```

```
int wc_curve25519_generic(
    int public_size,
    byte * pub,
    int private_size,
    const byte * priv,
    int basepoint_size,
```

```
    const byte * basepoint
)
```

This function performs a generic Curve25519 scalar multiplication with a custom basepoint. This allows computing scalar * basepoint for any basepoint, not just the standard generator.

Parameters:

- **public_size** Size of the output buffer (must be 32)
- **pub** Pointer to buffer to store the result
- **private_size** Size of the scalar (must be 32)
- **priv** Pointer to buffer containing the scalar
- **basepoint_size** Size of the basepoint (must be 32)
- **basepoint** Pointer to buffer containing the basepoint

See:

- wc_curve25519_shared_secret
- wc_curve25519_generic_blind

Return:

- 0 On successfully computing the result
- ECC_BAD_ARG_E If the sizes are invalid
- BAD_FUNC_ARG If any input parameters are NULL

Example

```
byte scalar[CURVE25519_KEYSIZE];
byte basepoint[CURVE25519_KEYSIZE];
byte result[CURVE25519_KEYSIZE];

// initialize scalar and basepoint
int ret = wc_curve25519_generic(sizeof(result), result,
                                sizeof(scalar), scalar,
                                sizeof(basepoint), basepoint);
if (ret != 0) {
    // error computing result
}
```

```
int wc_curve25519_generic_blind(
    int public_size,
    byte * pub,
    int private_size,
    const byte * priv,
    int basepoint_size,
    const byte * basepoint,
    WC_RNG * rng
)
```

This function performs a generic Curve25519 scalar multiplication with a custom basepoint and blinding to resist side-channel attacks.

Parameters:

- **public_size** Size of the output buffer (must be 32)
- **pub** Pointer to buffer to store the result
- **private_size** Size of the scalar (must be 32)
- **priv** Pointer to buffer containing the scalar
- **basepoint_size** Size of the basepoint (must be 32)
- **basepoint** Pointer to buffer containing the basepoint
- **rng** Pointer to initialized RNG for blinding

See:

- wc_curve25519_generic
- wc_curve25519_make_pub_blind

Return:

- 0 On successfully computing the result
- ECC_BAD_ARG_E If the sizes are invalid
- BAD_FUNC_ARG If any input parameters are NULL

Example

```
WC_RNG rng;
byte scalar[CURVE25519_KEYSIZE];
byte basepoint[CURVE25519_KEYSIZE];
byte result[CURVE25519_KEYSIZE];

wc_InitRng(&rng);
// initialize scalar and basepoint
int ret = wc_curve25519_generic_blind(sizeof(result), result,
                                      sizeof(scalar), scalar,
                                      sizeof(basepoint), basepoint,
                                      &rng);
```

```
int wc_curve25519_make_priv(
    WC_RNG * rng,
    int keysize,
    byte * priv
)
```

This function generates a Curve25519 private key using the given random number generator. This is a lower-level function that generates only the private key bytes.

Parameters:

- **rng** Pointer to initialized RNG

- **keysize** Size of the key to generate (must be 32)
- **priv** Pointer to buffer to store the private key

See:

- wc_curve25519_make_key
- wc_curve25519_make_pub

Return:

- 0 On successfully generating the private key
- ECC_BAD_ARG_E If keysize is invalid
- BAD_FUNC_ARG If any input parameters are NULL
- RNG_FAILURE_E If random number generation fails

Example

```
WC_RNG rng;
byte priv[CURVE25519_KEYSIZE];

wc_InitRng(&rng);
int ret = wc_curve25519_make_priv(&rng, sizeof(priv), priv);
if (ret != 0) {
    // error generating private key
}
```

```
int wc_curve25519_init_ex(
    curve25519_key * key,
    void * heap,
    int devId
)
```

This function initializes a Curve25519 key with extended parameters, allowing specification of custom heap and device ID for hardware acceleration.

Parameters:

- **key** Pointer to the curve25519_key structure to initialize
- **heap** Pointer to heap hint for memory allocation (can be NULL)
- **devId** Device ID for hardware acceleration (use INVALID_DEVID for software only)

See:

- wc_curve25519_init
- wc_curve25519_free

Return:

- 0 On successfully initializing the key

- BAD_FUNC_ARG If key is NULL

Example

```
curve25519_key key;
void* heap = NULL;
int devId = INVALID_DEVID;

int ret = wc_curve25519_init_ex(&key, heap, devId);
if (ret != 0) {
    // error initializing key
}
```

```
int wc_curve25519_set_rng(
    curve25519_key * key,
    WC_RNG * rng
)
```

This function sets the RNG to be used with a Curve25519 key. This is useful for operations that require randomness such as blinded scalar multiplication.

Parameters:

- **key** Pointer to the curve25519_key structure
- **rng** Pointer to initialized RNG

See:

- wc_curve25519_init
- wc_curve25519_make_key

Return:

- 0 On successfully setting the RNG
- BAD_FUNC_ARG If key or rng is NULL

Example

```
WC_RNG rng;
curve25519_key key;

wc_InitRng(&rng);
wc_curve25519_init(&key);
int ret = wc_curve25519_set_rng(&key, &rng);
if (ret != 0) {
    // error setting RNG
}
```

```
curve25519_key * wc_curve25519_new(
    void * heap,
    int devId,
    int * result_code
)
```

This function allocates and initializes a new Curve25519 key structure with extended parameters. The caller is responsible for freeing the key with wc_curve25519_delete. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.

Parameters:

- **heap** Pointer to heap hint for memory allocation (can be NULL)
- **devId** Device ID for hardware acceleration (use INVALID_DEVID for software only)
- **result_code** Pointer to store result code (0 on success)

See:

- wc_curve25519_delete
- wc_curve25519_init_ex

Return:

- Pointer to newly allocated curve25519_key on success
- NULL on failure

Note: This API is only available when WC_NO_CONSTRUCTORS is not defined. WC_NO_CONSTRUCTORS is automatically defined when WOLFSSL_NO_MALLOC is defined.

Example

```
int ret;
curve25519_key* key;

key = wc_curve25519_new(NULL, INVALID_DEVID, &ret);
if (key == NULL || ret != 0) {
    // error allocating key
}
// use key
wc_curve25519_delete(key, &key);
```

```
int wc_curve25519_delete(
    curve25519_key * key,
    curve25519_key ** key_p
)
```

This function frees a Curve25519 key structure that was allocated with wc_curve25519_new and sets the pointer to NULL. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.

Parameters:

- **key** Pointer to the curve25519_key structure to free
- **key_p** Pointer to the key pointer (will be set to NULL)

See:

- wc_curve25519_new
- wc_curve25519_free

Return:

- 0 On successfully freeing the key
- BAD_FUNC_ARG If key or key_p is NULL

Note: This API is only available when WC_NO_CONSTRUCTORS is not defined. WC_NO_CONSTRUCTORS is automatically defined when WOLFSSL_NO_MALLOC is defined.

Example

```
int ret;
curve25519_key* key;

key = wc_curve25519_new(NULL, INVALID_DEVID, &ret);
// use key
ret = wc_curve25519_delete(key, &key);
if (ret != 0) {
    // error freeing key
}
// key is now NULL
```

B.21 Algorithms - Curve448

B.20.2.28 function wc_curve25519_delete

B.21.1 Functions

	Name
int	**wc_Curve448PrivateKeyDecode**(const byte * input, word32 * inOutIdx, curve448_key * key, word32 inSz)Decodes Curve448 private key from DER format.
int	**wc_Curve448PublicKeyDecode**(const byte * input, word32 * inOutIdx, curve448_key * key, word32 inSz)Decodes Curve448 public key from DER format.
int	**wc_Curve448PrivateKeyToDer**(curve448_key * key, byte * output, word32 outLen)Encodes Curve448 private key to DER format.

	Name
int	**wc_Curve448PublicKeyToDer**(curve448_key * key, byte * output, word32 outLen)Encodes Curve448 public key to DER format.
int	**wc_curve448_make_key**(WC_RNG * rng, int keysize, curve448_key * key)This function generates a Curve448 key using the given random number generator, rng, of the size given (keysize), and stores it in the given curve448_key structure. It should be called after the key structure has been initialized through wc_curve448_init().
int	**wc_curve448_shared_secret**(curve448_key * private_key, curve448_key * public_key, byte * out, word32 * outlen)This function computes a shared secret key given a secret private key and a received public key. It stores the generated secret key in the buffer out and assigns the variable of the secret key to outlen. Only supports big endian.
int	**wc_curve448_shared_secret_ex**(curve448_key * private_key, curve448_key * public_key, byte * out, word32 * outlen, int endian)This function computes a shared secret key given a secret private key and a received public key. It stores the generated secret key in the buffer out and assigns the variable of the secret key to outlen. Supports both big and little endian.
int	**wc_curve448_init**(curve448_key * key)This function initializes a Curve448 key. It should be called before generating a key for the structure.
int	**wc_curve448_init_ex**(curve448_key * key, void * heap, int devId)This function initializes a Curve448 key with extended parameters, allowing specification of custom heap and device ID for hardware acceleration.
curve448_key *	**wc_curve448_new**(void * heap, int devId, int * result_code)This function allocates and initializes a new Curve448 key structure with extended parameters. The caller is responsible for freeing the key with wc_curve448_delete. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.

	Name
int	**wc_curve448_delete**(curve448_key * key, curve448_key ** key_p)This function frees a Curve448 key structure that was allocated with wc_curve448_new and sets the pointer to NULL. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.
void	**wc_curve448_free**(curve448_key * key)This function frees a Curve448 object.
int	**wc_curve448_import_private**(const byte * priv, word32 privSz, curve448_key * key)This function imports a curve448 private key only. (Big endian).
int	**wc_curve448_import_private_ex**(const byte * priv, word32 privSz, curve448_key * key, int endian)curve448 private key import only. (Big or Little endian).
int	**wc_curve448_import_private_raw**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, curve448_key * key)This function imports a public-private key pair into a curve448_key structure. Big endian only.
int	**wc_curve448_import_private_raw_ex**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, curve448_key * key, int endian)This function imports a public-private key pair into a curve448_key structure. Supports both big and little endian.
int	**wc_curve448_export_private_raw**(curve448_key * key, byte * out, word32 * outLen)This function exports a private key from a curve448_key structure and stores it in the given out buffer. It also sets outLen to be the size of the exported key. Big Endian only.
int	**wc_curve448_export_private_raw_ex**(curve448_key * key, byte * out, word32 * outLen, int endian)This function exports a private key from a curve448_key structure and stores it in the given out buffer. It also sets outLen to be the size of the exported key. Can specify whether it's big or little endian.
int	**wc_curve448_import_public**(const byte * in, word32 inLen, curve448_key * key)This function imports a public key from the given in buffer and stores it in the curve448_key structure.
int	**wc_curve448_import_public_ex**(const byte * in, word32 inLen, curve448_key * key, int endian)This function imports a public key from the given in buffer and stores it in the curve448_key structure.

	Name
int	**wc_curve448_check_public**(const byte * pub, word32 pubSz, int endian)This function checks that a public key buffer holds a valid Curve448 key value given the endian ordering.
int	**wc_curve448_export_public**(curve448_key * key, byte * out, word32 * outLen)This function exports a public key from the given key structure and stores the result in the out buffer. Big endian only.
int	**wc_curve448_export_public_ex**(curve448_key * key, byte * out, word32 * outLen, int endian)This function exports a public key from the given key structure and stores the result in the out buffer. Supports both big and little endian.
int	**wc_curve448_export_key_raw**(curve448_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz)This function exports a key pair from the given key structure and stores the result in the out buffer. Big endian only.
int	**wc_curve448_export_key_raw_ex**(curve448_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz, int endian)Export curve448 key pair. Big or little endian.
int	**wc_curve448_size**(curve448_key * key)This function returns the key size of the given key structure.
int	**wc_curve448_make_pub**(int public_size, byte * pub, int private_size, const byte * priv)This function generates a Curve448 public key from a given private key. It computes the public key by performing scalar multiplication of the base point with the private key.
int	**wc_curve448_generic**(int public_size, byte * pub, int private_size, const byte * priv, int basepoint_size, const byte * basepoint)This function performs a generic Curve448 scalar multiplication with a custom basepoint. This allows computing scalar * basepoint for any basepoint, not just the standard generator. As in wc_curve448_shared_secret_ex, an all_zero result is rejected (RFC 7748 section 6.2); build with WOLFSSL_NO_ECDHX_SHARED_ZERO_CHECK to get the raw scalar multiplication result instead.

B.21.2 Functions Documentation

```
int wc_Curve448PrivateKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    curve448_key * key,
```

```
    word32 inSz
)
```

Decodes Curve448 private key from DER format.

Parameters:

- **input** DER encoded Curve448 private key buffer
- **inOutIdx** Pointer to index in buffer
- **key** Curve448 key structure to store key
- **inSz** Size of input buffer

See: wc_Curve448PrivateKeyToDer

Return:

- 0 on success
- negative on error

Example

```
curve448_key key;
word32 idx = 0;
int ret = wc_Curve448PrivateKeyDecode(derBuf, &idx, &key,
                                      derSz);
```

```
int wc_Curve448PublicKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    curve448_key * key,
    word32 inSz
)
```

Decodes Curve448 public key from DER format.

Parameters:

- **input** DER encoded Curve448 public key buffer
- **inOutIdx** Pointer to index in buffer
- **key** Curve448 key structure to store key
- **inSz** Size of input buffer

See: wc_Curve448PublicKeyToDer

Return:

- 0 on success
- negative on error

Example

```
curve448_key key;
word32 idx = 0;
int ret = wc_Curve448PublicKeyDecode(derBuf, &idx, &key,
                                     derSz);
```

```
int wc_Curve448PrivateKeyToDer(
    curve448_key * key,
    byte * output,
    word32 outLen
)
```

Encodes Curve448 private key to DER format.

Parameters:

- **key** Curve448 key structure with private key
- **output** Buffer for DER encoded private key
- **outLen** Size of output buffer

See: wc_Curve448PrivateKeyDecode

Return:

- Size on success
- negative on error

Example

```
curve448_key key;
byte der[1024];
int derSz = wc_Curve448PrivateKeyToDer(&key, der,
                                       sizeof(der));
```

```
int wc_Curve448PublicKeyToDer(
    curve448_key * key,
    byte * output,
    word32 outLen
)
```

Encodes Curve448 public key to DER format.

Parameters:

- **key** Curve448 key structure with public key
- **output** Buffer for DER encoded public key
- **outLen** Size of output buffer

See: wc_Curve448PublicKeyDecode

Return:

- Size on success
- negative on error

Example

```
curve448_key key;
byte der[1024];
int derSz = wc_Curve448PublicKeyToDer(&key, der,
                                      sizeof(der));
```

```
int wc_curve448_make_key(
    WC_RNG * rng,
    int keysize,
    curve448_key * key
)
```

This function generates a Curve448 key using the given random number generator, rng, of the size given (keysize), and stores it in the given curve448_key structure. It should be called after the key structure has been initialized through wc_curve448_init().

Parameters:

- **rng** Pointer to the RNG object used to generate the ecc key.
- **keysize** Size of the key to generate. Must be 56 bytes for curve448.
- **key** Pointer to the curve448_key structure in which to store the generated key.

See: wc_curve448_init

Return:

- 0 Returned on successfully generating the key and and storing it in the given curve448_key structure.
- ECC_BAD_ARG_E Returned if the input keysize does not correspond to the keysize for a curve448 key (56 bytes).
- RNG_FAILURE_E Returned if the rng internal status is not DRBG_OK or if there is in generating the next random block with rng.
- BAD_FUNC_ARG Returned if any of the input parameters passed in are NULL.

Example

```
int ret;

curve448_key key;
wc_curve448_init(&key); // initialize key
WC_RNG rng;
wc_InitRng(&rng); // initialize random number generator
```

```
ret = wc_curve448_make_key(&rng, 56, &key);
if (ret != 0) {
    // error making Curve448 key
}
```

```
int wc_curve448_shared_secret(
    curve448_key * private_key,
    curve448_key * public_key,
    byte * out,
    word32 * outlen
)
```

This function computes a shared secret key given a secret private key and a received public key. It stores the generated secret key in the buffer out and assigns the variable of the secret key to outlen. Only supports big endian.

Parameters:

- **private_key** Pointer to the curve448_key structure initialized with the user's private key.
- **public_key** Pointer to the curve448_key structure containing the received public key.
- **out** Pointer to a buffer in which to store the 56 byte computed secret key.
- **outlen** Pointer in which to store the length written to the output buffer.

See:

- wc_curve448_init
- wc_curve448_make_key
- wc_curve448_shared_secret_ex

Return:

- 0 Returned on successfully computing a shared secret key
- BAD_FUNC_ARG Returned if any of the input parameters passed in are NULL

Example

```
int ret;

byte sharedKey[56];
word32 keySz;
curve448_key privKey, pubKey;
// initialize both keys

ret = wc_curve448_shared_secret(&privKey, &pubKey, sharedKey, &keySz);
if (ret != 0) {
    // error generating shared key
}
```

```
int wc_curve448_shared_secret_ex(
    curve448_key * private_key,
    curve448_key * public_key,
    byte * out,
    word32 * outlen,
    int endian
)
```

This function computes a shared secret key given a secret private key and a received public key. It stores the generated secret key in the buffer out and assigns the variable of the secret key to outlen. Supports both big and little endian.

Parameters:

- **private_key** Pointer to the curve448_key structure initialized with the user's private key.
- **public_key** Pointer to the curve448_key structure containing the received public key.
- **out** Pointer to a buffer in which to store the 56 byte computed secret key.
- **outlen** Pointer in which to store the length written to the output buffer.
- **endian** EC448_BIG_ENDIAN or EC448_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve448_init
- wc_curve448_make_key
- wc_curve448_shared_secret

Return:

- 0 Returned on successfully computing a shared secret key.
- BAD_FUNC_ARG Returned if any of the input parameters passed in are NULL.

Example

```
int ret;

byte sharedKey[56];
word32 keySz;

curve448_key privKey, pubKey;
// initialize both keys

ret = wc_curve448_shared_secret_ex(&privKey, &pubKey, sharedKey, &keySz,
        EC448_BIG_ENDIAN);
if (ret != 0) {
    // error generating shared key
}
```

```
int wc_curve448_init(
    curve448_key * key
)
```

This function initializes a Curve448 key. It should be called before generating a key for the structure.

Parameters:

- **key** Pointer to the curve448_key structure to initialize.

See: wc_curve448_make_key

Return:

- 0 Returned on successfully initializing the curve448_key structure.
- BAD_FUNC_ARG Returned when key is NULL.

Example

```
curve448_key key;
wc_curve448_init(&key); // initialize key
// make key and proceed to encryption
```

```
int wc_curve448_init_ex(
    curve448_key * key,
    void * heap,
    int devId
)
```

This function initializes a Curve448 key with extended parameters, allowing specification of custom heap and device ID for hardware acceleration.

Parameters:

- **key** Pointer to the curve448_key structure to initialize.
- **heap** Pointer to heap hint for memory allocation (can be NULL)
- **devId** Device ID for hardware acceleration (use INVALID_DEVID for software only)

See:

- wc_curve448_init
- wc_curve448_free

Return:

- 0 On successfully initializing the key
- BAD_FUNC_ARG If key is NULL

Example

```
curve448_key key;
void* heap = NULL;
int devId = INVALID_DEVID;
```

```
int ret = wc_curve448_init_ex(&key, heap, devId);
if (ret != 0) {
    // error initializing key
}
```

```
curve448_key * wc_curve448_new(
    void * heap,
    int devId,
    int * result_code
)
```

This function allocates and initializes a new Curve448 key structure with extended parameters. The caller is responsible for freeing the key with wc_curve448_delete. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.

Parameters:

- **heap** Pointer to heap hint for memory allocation (can be NULL)
- **devId** Device ID for hardware acceleration (use INVALID_DEVID for software only)
- **result_code** Pointer to store result code (0 on success)

See:

- wc_curve448_delete
- wc_curve448_init_ex

Return:

- Pointer to newly allocated curve448_key on success
- NULL on failure

Note: This API is only available when WC_NO_CONSTRUCTORS is not defined. WC_NO_CONSTRUCTORS is automatically defined when WOLFSSL_NO_MALLOC is defined.

Example

```
int ret;
curve448_key* key;

key = wc_curve448_new(NULL, INVALID_DEVID, &ret);
if (key == NULL || ret != 0) {
    // error allocating key
}
// use key
wc_curve448_delete(key, &key);
```

```
int wc_curve448_delete(
    curve448_key * key,
    curve448_key ** key_p
)
```

This function frees a Curve448 key structure that was allocated with wc_curve448_new and sets the pointer to NULL. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.

Parameters:

- **key** Pointer to the curve448_key structure to free.
- **key_p** Pointer to the key pointer to set to NULL (can be NULL)

See:

- wc_curve448_new
- wc_curve448_free

Return:

- 0 On successfully freeing the key
- BAD_FUNC_ARG If key is NULL

Note: This API is only available when WC_NO_CONSTRUCTORS is not defined. WC_NO_CONSTRUCTORS is automatically defined when WOLFSSL_NO_MALLOC is defined.

Example

```
int ret;
curve448_key* key = wc_curve448_new(NULL, INVALID_DEVID, &ret);
// use key
ret = wc_curve448_delete(key, &key);
```

```
void wc_curve448_free(
    curve448_key * key
)
```

This function frees a Curve448 object.

Parameters:

- **key** Pointer to the key object to free.

See:

- wc_curve448_init
- wc_curve448_make_key

Example

```
curve448_key privKey;
// initialize key, use it to generate shared secret key
wc_curve448_free(&privKey);
```

```
int wc_curve448_import_private(
    const byte * priv,
    word32 privSz,
    curve448_key * key
)
```

This function imports a curve448 private key only. (Big endian).

Parameters:

- **priv** Pointer to a buffer containing the private key to import.
- **privSz** Length of the private key to import.
- **key** Pointer to the structure in which to store the imported key.

See:

- wc_curve448_import_private_ex
- wc_curve448_size

Return:

- 0 Returned on successfully importing private key.
- BAD_FUNC_ARG Returns if key or priv is null.
- ECC_BAD_ARG_E Returns if privSz is not equal to CURVE448_KEY_SIZE.

Example

```
int ret;

byte priv[] = { Contents of private key };
curve448_key key;
wc_curve448_init(&key);

ret = wc_curve448_import_private(priv, sizeof(priv), &key);
if (ret != 0) {
    // error importing key
}
```

```
int wc_curve448_import_private_ex(
    const byte * priv,
    word32 privSz,
```

```
    curve448_key * key,
    int endian
)
```

curve448 private key import only. (Big or Little endian).

Parameters:

- **priv** Pointer to a buffer containing the private key to import.
- **privSz** Length of the private key to import.
- **key** Pointer to the structure in which to store the imported key.
- **endian** EC448_BIG_ENDIAN or EC448_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve448_import_private
- wc_curve448_size

Return:

- 0 Returned on successfully importing private key.
- BAD_FUNC_ARG Returns if key or priv is null.
- ECC_BAD_ARG_E Returns if privSz is not equal to CURVE448_KEY_SIZE.

Example

```
int ret;

byte priv[] = { // Contents of private key };
curve448_key key;
wc_curve448_init(&key);

ret = wc_curve448_import_private_ex(priv, sizeof(priv), &key,
        EC448_BIG_ENDIAN);
if (ret != 0) {
    // error importing key
}
```

```
int wc_curve448_import_private_raw(
    const byte * priv,
    word32 privSz,
    const byte * pub,
    word32 pubSz,
    curve448_key * key
)
```

This function imports a public-private key pair into a curve448_key structure. Big endian only.

Parameters:

- **priv** Pointer to a buffer containing the private key to import.
- **privSz** Length of the private key to import.
- **pub** Pointer to a buffer containing the public key to import.
- **pubSz** Length of the public key to import.
- **key** Pointer to the structure in which to store the imported keys

See:

- wc_curve448_init
- wc_curve448_make_key
- wc_curve448_import_public
- wc_curve448_export_private_raw

Return:

- 0 Returned on importing into the curve448_key structure.
- BAD_FUNC_ARG Returns if any of the input parameters are null.
- ECC_BAD_ARG_E Returned if the input key's key size does not match the public or private key sizes.

Example

```
int ret;

byte priv[56];
byte pub[56];
// initialize with public and private keys
curve448_key key;

wc_curve448_init(&key);
// initialize key

ret = wc_curve448_import_private_raw(&priv, sizeof(priv), pub, sizeof(pub),
        &key);
if (ret != 0) {
    // error importing keys
}
```

```
int wc_curve448_import_private_raw_ex(
    const byte * priv,
    word32 privSz,
    const byte * pub,
    word32 pubSz,
    curve448_key * key,
    int endian
)
```

This function imports a public-private key pair into a curve448_key structure. Supports both big and little endian.

Parameters:

- **priv** Pointer to a buffer containing the private key to import.
- **privSz** Length of the private key to import.
- **pub** Pointer to a buffer containing the public key to import.
- **pubSz** Length of the public key to import.
- **key** Pointer to the structure in which to store the imported keys.
- **endian** EC448_BIG_ENDIAN or EC448_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve448_init
- wc_curve448_make_key
- wc_curve448_import_public
- wc_curve448_export_private_raw
- wc_curve448_import_private_raw

Return:

- 0 Returned on importing into the curve448_key structure.
- BAD_FUNC_ARG Returns if any of the input parameters are null.
- ECC_BAD_ARG_E Returned if the input key's key size does not match the public or private key sizes.

Example

```
int ret;

byte priv[56];
byte pub[56];
// initialize with public and private keys
curve448_key key;

wc_curve448_init(&key);
// initialize key

ret = wc_curve448_import_private_raw_ex(&priv, sizeof(priv), pub,
        sizeof(pub), &key, EC448_BIG_ENDIAN);
if (ret != 0) {
    // error importing keys
}
```

```
int wc_curve448_export_private_raw(
    curve448_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports a private key from a curve448_key structure and stores it in the given out buffer. It also sets outLen to be the size of the exported key. Big Endian only.

Parameters:

- **key** Pointer to the structure from which to export the key.
- **out** Pointer to the buffer in which to store the exported key.
- **outLen** On in, is the size of the out in bytes. On out, will store the bytes written to the output buffer.

See:

- wc_curve448_init
- wc_curve448_make_key
- wc_curve448_import_private_raw
- wc_curve448_export_private_raw_ex

Return:

- 0 Returned on successfully exporting the private key from the curve448_key structure.
- BAD_FUNC_ARG Returned if any input parameters are NULL.
- ECC_BAD_ARG_E Returned if wc_curve448_size() is not equal to key.

Example

```
int ret;
byte priv[56];
int privSz;

curve448_key key;
// initialize and make key

ret = wc_curve448_export_private_raw(&key, priv, &privSz);
if (ret != 0) {
    // error exporting key
}
```

```
int wc_curve448_export_private_raw_ex(
    curve448_key * key,
    byte * out,
    word32 * outLen,
    int endian
)
```

This function exports a private key from a curve448_key structure and stores it in the given out buffer. It also sets outLen to be the size of the exported key. Can specify whether it's big or little endian.

Parameters:

- **key** Pointer to the structure from which to export the key.
- **out** Pointer to the buffer in which to store the exported key.
- **outLen** On in, is the size of the out in bytes. On out, will store the bytes written to the output buffer.
- **endian** EC448_BIG_ENDIAN or EC448_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve448_init
- wc_curve448_make_key
- wc_curve448_import_private_raw
- wc_curve448_export_private_raw
- wc_curve448_size

Return:

- 0 Returned on successfully exporting the private key from the curve448_key structure.
- BAD_FUNC_ARG Returned if any input parameters are NULL.
- ECC_BAD_ARG_E Returned if wc_curve448_size() is not equal to key.

Example

```
int ret;

byte priv[56];
int privSz;
curve448_key key;
// initialize and make key
ret = wc_curve448_export_private_raw_ex(&key, priv, &privSz,
        EC448_BIG_ENDIAN);
if (ret != 0) {
    // error exporting key
}
```

```
int wc_curve448_import_public(
    const byte * in,
    word32 inLen,
    curve448_key * key
)
```

This function imports a public key from the given in buffer and stores it in the curve448_key structure.

Parameters:

- **in** Pointer to the buffer containing the public key to import.
- **inLen** Length of the public key to import.
- **key** Pointer to the curve448_key structure in which to store the key.

See:

- wc_curve448_init
- wc_curve448_export_public
- wc_curve448_import_private_raw
- wc_curve448_import_public_ex
- wc_curve448_check_public

- wc_curve448_size

Return:

- 0 Returned on successfully importing the public key into the curve448_key structure.
- ECC_BAD_ARG_E Returned if the inLen parameter does not match the key size of the key structure.
- BAD_FUNC_ARG Returned if any of the input parameters are NULL.

Example

```
int ret;

byte pub[56];
// initialize pub with public key

curve448_key key;
// initialize key

ret = wc_curve448_import_public(pub,sizeof(pub), &key);
if (ret != 0) {
    // error importing key
}
```

```
int wc_curve448_import_public_ex(
    const byte * in,
    word32 inLen,
    curve448_key * key,
    int endian
)
```

This function imports a public key from the given in buffer and stores it in the curve448_key structure.

Parameters:

- **in** Pointer to the buffer containing the public key to import.
- **inLen** Length of the public key to import.
- **key** Pointer to the curve448_key structure in which to store the key.
- **endian** EC448_BIG_ENDIAN or EC448_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve448_init
- wc_curve448_export_public
- wc_curve448_import_private_raw
- wc_curve448_import_public
- wc_curve448_check_public
- wc_curve448_size

Return:

- 0 Returned on successfully importing the public key into the curve448_key structure.
- ECC_BAD_ARG_E Returned if the inLen parameter does not match the key size of the key structure.
- BAD_FUNC_ARG Returned if any of the input parameters are NULL.

Example

```
int ret;

byte pub[56];
// initialize pub with public key
curve448_key key;
// initialize key

ret = wc_curve448_import_public_ex(pub, sizeof(pub), &key,
        EC448_BIG_ENDIAN);
if (ret != 0) {
    // error importing key
}
```

```
int wc_curve448_check_public(
    const byte * pub,
    word32 pubSz,
    int endian
)
```

This function checks that a public key buffer holds a valid Curve448 key value given the endian ordering.

Parameters:

- **pub** Pointer to the buffer containing the public key to check.
- **pubSz** Length of the public key to check.
- **endian** EC448_BIG_ENDIAN or EC448_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve448_init
- wc_curve448_import_public
- wc_curve448_import_public_ex
- wc_curve448_size

Return:

- 0 Returned when the public key value is valid.
- ECC_BAD_ARG_E Returned if the public key value is not valid.
- BAD_FUNC_ARG Returned if any of the input parameters are NULL.

Example

```
int ret;

byte pub[] = { Contents of public key };

ret = wc_curve448_check_public_ex(pub, sizeof(pub), EC448_BIG_ENDIAN);
if (ret != 0) {
    // error importing key
}
```

```
int wc_curve448_export_public(
    curve448_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports a public key from the given key structure and stores the result in the out buffer. Big endian only.

Parameters:

- **key** Pointer to the curve448_key structure in from which to export the key.
- **out** Pointer to the buffer in which to store the public key.
- **outLen** On in, is the size of the out in bytes. On out, will store the bytes written to the output buffer.

See:

- wc_curve448_init
- wc_curve448_export_private_raw
- wc_curve448_import_public

Return:

- 0 Returned on successfully exporting the public key from the curve448_key structure.
- ECC_BAD_ARG_E Returned if outLen is less than CURVE448_PUB_KEY_SIZE, or if neither the public nor the private key has been set.
- BAD_FUNC_ARG Returned if any of the input parameters are NULL.

Example

```
int ret;

byte pub[56];
int pubSz;

curve448_key key;
// initialize and make key
```

```
ret = wc_curve448_export_public(&key, pub, &pubSz);
if (ret != 0) {
    // error exporting key
}
```

```
int wc_curve448_export_public_ex(
    curve448_key * key,
    byte * out,
    word32 * outLen,
    int endian
)
```

This function exports a public key from the given key structure and stores the result in the out buffer. Supports both big and little endian.

Parameters:

- **key** Pointer to the curve448_key structure in from which to export the key.
- **out** Pointer to the buffer in which to store the public key.
- **outLen** On in, is the size of the out in bytes. On out, will store the bytes written to the output buffer.
- **endian** EC448_BIG_ENDIAN or EC448_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve448_init
- wc_curve448_export_private_raw
- wc_curve448_import_public

Return:

- 0 Returned on successfully exporting the public key from the curve448_key structure.
- ECC_BAD_ARG_E Returned if outLen is less than CURVE448_PUB_KEY_SIZE, or if neither the public nor the private key has been set.
- BAD_FUNC_ARG Returned if any of the input parameters are NULL.

Example

```
int ret;

byte pub[56];
int pubSz;

curve448_key key;
// initialize and make key

ret = wc_curve448_export_public_ex(&key, pub, &pubSz, EC448_BIG_ENDIAN);
if (ret != 0) {
```

```
    // error exporting key
}



int wc_curve448_export_key_raw(
    curve448_key * key,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz
)
```

This function exports a key pair from the given key structure and stores the result in the out buffer. Big endian only.

Parameters:

- **key** Pointer to the curve448_key structure in from which to export the key pair.
- **priv** Pointer to the buffer in which to store the private key.
- **privSz** On in, is the size of the priv buffer in bytes. On out, will store the bytes written to the priv buffer.
- **pub** Pointer to the buffer in which to store the public key.
- **pubSz** On in, is the size of the pub buffer in bytes. On out, will store the bytes written to the pub buffer.

See:

- wc_curve448_export_key_raw_ex
- wc_curve448_export_private_raw

Return:

- 0 Returned on successfully exporting the key pair from the curve448_key structure.
- BAD_FUNC_ARG Returned if any input parameters are NULL.
- ECC_BAD_ARG_E Returned if privSz is less than CURVE448_KEY_SIZE or pubSz is less than CURVE448_PUB_KEY_SIZE.

Example

```
int ret;

byte pub[56];
byte priv[56];
int pubSz;
int privSz;

curve448_key key;
// initialize and make key

ret = wc_curve448_export_key_raw(&key, priv, &privSz, pub, &pubSz);
if (ret != 0) {
```

```
    // error exporting key
}



int wc_curve448_export_key_raw_ex(
    curve448_key * key,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz,
    int endian
)
```

Export curve448 key pair. Big or little endian.

Parameters:

- **key** Pointer to the curve448_key structure in from which to export the key pair.
- **priv** Pointer to the buffer in which to store the private key.
- **privSz** On in, is the size of the priv buffer in bytes. On out, will store the bytes written to the priv buffer.
- **pub** Pointer to the buffer in which to store the public key.
- **pubSz** On in, is the size of the pub buffer in bytes. On out, will store the bytes written to the pub buffer.
- **endian** EC448_BIG_ENDIAN or EC448_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve448_export_key_raw
- wc_curve448_export_private_raw_ex
- wc_curve448_export_public_ex

Return:

- 0 Success
- BAD_FUNC_ARG Returned if any input parameters are NULL.
- ECC_BAD_ARG_E Returned if privSz is less than CURVE448_KEY_SIZE or pubSz is less than CURVE448_PUB_KEY_SIZE.

This function exports a key pair from the given key structure and stores the result in the out buffer. Big or little endian.

Example

```
int ret;

byte pub[56];
byte priv[56];
int pubSz;
int privSz;

curve448_key key;
```

```
// initialize and make key

ret = wc_curve448_export_key_raw_ex(&key,priv, &privSz, pub, &pubSz,
        EC448_BIG_ENDIAN);
if (ret != 0) {
    // error exporting key
}
```

```
int wc_curve448_size(
    curve448_key * key
)
```

This function returns the key size of the given key structure.

Parameters:

- **key** Pointer to the curve448_key structure in for which to determine the key size.

See:

- wc_curve448_init
- wc_curve448_make_key

Return:

- Success Given a valid, initialized curve448_key structure, returns the size of the key.
- 0 Returned if key is NULL.

Example

```
int keySz;

curve448_key key;
// initialize and make key

keySz = wc_curve448_size(&key);
```

```
int wc_curve448_make_pub(
    int public_size,
    byte * pub,
    int private_size,
    const byte * priv
)
```

This function generates a Curve448 public key from a given private key. It computes the public key by performing scalar multiplication of the base point with the private key.

Parameters:

- **public_size** size of the public key buffer (must be 56 bytes)
- **pub** pointer to buffer to store the generated public key
- **private_size** size of the private key (must be 56 bytes)
- **priv** pointer to the private key buffer

See:

- wc_curve448_make_key
- wc_curve448_import_private

Return:

- 0 On success.
- ECC_BAD_ARG_E If public_size is not CURVE448_PUB_KEY_SIZE or if private_size is not CURVE448_KEY_SIZE.
- BAD_FUNC_ARG If pub or priv is NULL.

Example

```
byte priv[CURVE448_KEY_SIZE] = { }; // private key
byte pub[CURVE448_PUB_KEY_SIZE];

int ret = wc_curve448_make_pub(CURVE448_PUB_KEY_SIZE, pub,
                               CURVE448_KEY_SIZE, priv);
if (ret != 0) {
    // error generating public key
}
```

```
int wc_curve448_generic(
    int public_size,
    byte * pub,
    int private_size,
    const byte * priv,
    int basepoint_size,
    const byte * basepoint
)
```

This function performs a generic Curve448 scalar multiplication with a custom basepoint. This allows computing scalar * basepoint for any basepoint, not just the standard generator. As in wc_curve448_shared_secret_ex, an all-zero result is rejected (RFC 7748 section 6.2); build with WOLFSSL_NO_ECDHX_SHARED_ZERO_CHECK to get the raw scalar multiplication result instead.

Parameters:

- **public_size** Size of the output buffer (must be 56)
- **pub** Pointer to buffer to store the result
- **private_size** Size of the scalar (must be 56)
- **priv** Pointer to buffer containing the scalar
- **basepoint_size** Size of the basepoint (must be 56)
- **basepoint** Pointer to buffer containing the basepoint

See:

- wc_curve448_make_pub
- wc_curve448_shared_secret

Return:

- 0 On successfully computing the result
- ECC_BAD_ARG_E If any input parameter is NULL or a size is invalid
- ECC_OUT_OF_RANGE_E If the result is all-zero, i.e. the basepoint was of small order

Example

```
byte scalar[CURVE448_KEY_SIZE];
byte basepoint[CURVE448_KEY_SIZE];
byte result[CURVE448_PUB_KEY_SIZE];

// initialize scalar and basepoint
int ret = wc_curve448_generic(sizeof(result), result,
                              sizeof(scalar), scalar,
                              sizeof(basepoint), basepoint);
if (ret != 0) {
    // error computing result
}
```

B.22 Algorithms - DSA

B.21.2.28 function wc_curve448_generic

B.22.1 Functions

	Name
int	**wc_DsaParamsDecode**(const byte * input, word32 * inOutIdx, DsaKey * key, word32 inSz)Decodes DSA parameters from DER format.
int	**wc_DsaKeyToParamsDer**(DsaKey * key, byte * output, word32 outLen)Encodes DSA parameters to DER format.
int	**wc_DsaKeyToParamsDer_ex**(DsaKey * key, byte * output, word32 * outLen)Encodes DSA parameters to DER with size output.
int	**wc_InitDsaKey**(DsaKey * key)This function initializes a DsaKey object in order to use it for authentication via the Digital Signature Algorithm (DSA).
void	**wc_FreeDsaKey**(DsaKey * key)This function frees a DsaKey object after it has been used.

	Name
int	**wc_DsaSign**(const byte * digest, byte * out, DsaKey * key, WC_RNG * rng)This function signs the input digest and stores the result in the output buffer, out.
int	**wc_DsaVerify**(const byte * digest, const byte * sig, DsaKey * key, int * answer)This function verifies the signature of a digest, given a private key. It stores whether the key properly verifies in the answer parameter, with 1 corresponding to a successful verification, and 0 corresponding to failed verification.
int	**wc_DsaCheckPubKey**(DsaKey * key)This function validates DSA domain parameters and the public key contained in the given DsaKey. It performs the following checks (a subset of the requirements in FIPS 186_4 and NIST SP 800_89): p > 1 and q > 1; q divides (p _ 1); 1 < g < p; 1 < y < p; g^q mod p == 1 (so the multiplicative order of g divides q); and y^q mod p == 1 (so the multiplicative order of y divides q). The verify routines (wc_DsaVerify, wc_DsaVerify_ex) call this internally before any signature math runs, but the function is also exposed so callers can validate keys at import or decode time.
int	**wc_DsaPublicKeyDecode**(const byte * input, word32 * inOutIdx, DsaKey * key, word32 inSz)This function decodes a DER formatted certificate buffer containing a DSA public key, and stores the key in the given DsaKey structure. It also sets the inOutIdx parameter according to the length of the input read.
int	**wc_DsaPrivateKeyDecode**(const byte * input, word32 * inOutIdx, DsaKey * key, word32 inSz)This function decodes a DER formatted certificate buffer containing a DSA private key, and stores the key in the given DsaKey structure. It also sets the inOutIdx parameter according to the length of the input read.
int	**wc_DsaKeyToDer**(DsaKey * key, byte * output, word32 outLen)Convert DsaKey key to DER format, write to output (outLen), return bytes written.
int	**wc_MakeDsaKey**(WC_RNG * rng, DsaKey * dsa)Create a DSA key.
int	**wc_MakeDsaParameters**(WC_RNG * rng, int modulus_size, DsaKey * dsa)FIPS 186_4 defines valid for modulus_size values as (1024, 160) (2048, 256) (3072, 256)
int	**wc_InitDsaKey_h**(DsaKey * key, void * h)Initializes DSA key with heap hint.

	Name
int	**wc_DsaSign_ex**(const byte * digest, word32 digestSz, byte * out, DsaKey * key, WC_RNG * rng)Signs digest with extended parameters.
int	**wc_DsaVerify_ex**(const byte * digest, word32 digestSz, const byte * sig, DsaKey * key, int * answer)Verifies signature with extended parameters.
int	**wc_SetDsaPublicKey**(byte * output, DsaKey * key, int outLen, int with_header)Sets DSA public key in output buffer.
int	**wc_DsaKeyToPublicDer**(DsaKey * key, byte * output, word32 outLen)Converts DSA key to public DER format.
int	**wc_DsaImportParamsRaw**(DsaKey * dsa, const char * p, const char * q, const char * g)Imports DSA parameters from raw format. The parameters p, q, and g must be provided as ASCII hexadecimal strings (without 0x prefix). These represent the DSA domain parameters: p is the prime modulus, q is the prime divisor (subgroup order), and g is the generator.
int	**wc_DsaImportParamsRawCheck**(DsaKey * dsa, const char * p, const char * q, const char * g, int trusted, WC_RNG * rng)Imports DSA parameters from raw format with optional validation. The parameters p, q, and g must be provided as ASCII hexadecimal strings (without 0x prefix). The trusted parameter controls whether the prime p is validated: when trusted=1, prime checking is skipped (use when parameters come from a trusted source); when trusted=0, performs full primality testing on p (recommended for untrusted sources).
int	**wc_DsaExportParamsRaw**(DsaKey * dsa, byte * p, word32 * pSz, byte * q, word32 * qSz, byte * g, word32 * gSz)Exports DSA parameters to raw format.
int	**wc_DsaExportKeyRaw**(DsaKey * dsa, byte * x, word32 * xSz, byte * y, word32 * ySz)Exports DSA key to raw format.

B.22.2 Functions Documentation

```
int wc_DsaParamsDecode(
    const byte * input,
    word32 * inOutIdx,
    DsaKey * key,
    word32 inSz
)
```

Decodes DSA parameters from DER format.

Parameters:

- **input** DER encoded DSA parameters buffer
- **inOutIdx** Pointer to index in buffer
- **key** DSA key structure to store parameters
- **inSz** Size of input buffer

See: wc_DsaKeyToParamsDer

Return:

- 0 on success
- negative on error

Example

```
DsaKey key;
word32 idx = 0;
int ret = wc_DsaParamsDecode(derBuf, &idx, &key, derSz);
```

```
int wc_DsaKeyToParamsDer(
    DsaKey * key,
    byte * output,
    word32 outLen
)
```

Encodes DSA parameters to DER format.

Parameters:

- **key** DSA key structure with parameters
- **output** Buffer for DER encoded parameters
- **outLen** Size of output buffer

See: wc_DsaParamsDecode

Return:

- Size on success
- negative on error

Example

```
DsaKey key;
byte der[1024];
int derSz = wc_DsaKeyToParamsDer(&key, der, sizeof(der));
```

```
int wc_DsaKeyToParamsDer_ex(
    DsaKey * key,
    byte * output,
    word32 * outLen
)
```

Encodes DSA parameters to DER with size output.

Parameters:

- **key** DSA key structure with parameters
- **output** Buffer for DER encoded parameters
- **outLen** Pointer to buffer size (in/out)

See: wc_DsaKeyToParamsDer

Return:

- 0 on success
- negative on error

Example

```
DsaKey key;
byte der[1024];
word32 derSz = sizeof(der);
int ret = wc_DsaKeyToParamsDer_ex(&key, der, &derSz);
```

```
int wc_InitDsaKey(
    DsaKey * key
)
```

This function initializes a DsaKey object in order to use it for authentication via the Digital Signature Algorithm (DSA).

Parameters:

- **key** pointer to the DsaKey structure to initialize

See: wc_FreeDsaKey

Return:

- 0 Returned on success.
- BAD_FUNC_ARG Returned if a NULL key is passed in.

Example

```
DsaKey key;
int ret;
ret = wc_InitDsaKey(&key); // initialize DSA key
```

```
void wc_FreeDsaKey(
    DsaKey * key
)
```

This function frees a DsaKey object after it has been used.

Parameters:

- **key** pointer to the DsaKey structure to free

See: wc_FreeDsaKey

Return: none No returns.

Example

```
DsaKey key;
// initialize key, use for authentication
...
wc_FreeDsaKey(&key); // free DSA key
```

```
int wc_DsaSign(
    const byte * digest,
    byte * out,
    DsaKey * key,
    WC_RNG * rng
)
```

This function signs the input digest and stores the result in the output buffer, out.

Parameters:

- **digest** pointer to the hash to sign
- **out** pointer to the buffer in which to store the signature
- **key** pointer to the initialized DsaKey structure with which to generate the signature
- **rng** pointer to an initialized RNG to use with the signature generation

See: wc_DsaVerify

Return:

- 0 Returned on successfully signing the input digest
- MP_INIT_E may be returned if there is an error in processing the DSA signature.
- MP_READ_E may be returned if there is an error in processing the DSA signature.
- MP_CMP_E may be returned if there is an error in processing the DSA signature.
- MP_INVMOD_E may be returned if there is an error in processing the DSA signature.
- MP_EXPTMOD_E may be returned if there is an error in processing the DSA signature.
- MP_MOD_E may be returned if there is an error in processing the DSA signature.
- MP_MUL_E may be returned if there is an error in processing the DSA signature.

- MP_ADD_E may be returned if there is an error in processing the DSA signature.
- MP_MULMOD_E may be returned if there is an error in processing the DSA signature.
- MP_TO_E may be returned if there is an error in processing the DSA signature.
- MP_MEM may be returned if there is an error in processing the DSA signature.

Example

```
DsaKey key;
// initialize DSA key, load private Key
int ret;
WC_RNG rng;
wc_InitRng(&rng);
byte hash[] = { // initialize with hash digest };
byte signature[40]; // signature will be 40 bytes (320 bits)

ret = wc_DsaSign(hash, signature, &key, &rng);
if (ret != 0) {
    // error generating DSA signature
}
```

```
int wc_DsaVerify(
    const byte * digest,
    const byte * sig,
    DsaKey * key,
    int * answer
)
```

This function verifies the signature of a digest, given a private key. It stores whether the key properly verifies in the answer parameter, with 1 corresponding to a successful verification, and 0 corresponding to failed verification.

Parameters:

- **digest** pointer to the digest containing the subject of the signature
- **sig** pointer to the buffer containing the signature to verify
- **key** pointer to the initialized DsaKey structure with which to verify the signature
- **answer** pointer to an integer which will store whether the verification was successful

See: wc_DsaSign

Return:

- 0 Returned on successfully processing the verify request. Note: this does not mean that the signature is verified, only that the function succeeded
- MP_INIT_E may be returned if there is an error in processing the DSA signature.
- MP_READ_E may be returned if there is an error in processing the DSA signature.
- MP_CMP_E may be returned if there is an error in processing the DSA signature.
- MP_INVMOD_E may be returned if there is an error in processing the DSA signature.
- MP_EXPTMOD_E may be returned if there is an error in processing the DSA signature.
- MP_MOD_E may be returned if there is an error in processing the DSA signature.
- MP_MUL_E may be returned if there is an error in processing the DSA signature.

- MP_ADD_E may be returned if there is an error in processing the DSA signature.
- MP_MULMOD_E may be returned if there is an error in processing the DSA signature.
- MP_TO_E may be returned if there is an error in processing the DSA signature.
- MP_MEM may be returned if there is an error in processing the DSA signature.

Example

```
DsaKey key;
// initialize DSA key, load public Key

int ret;
int verified;
byte hash[] = { // initialize with hash digest };
byte signature[] = { // initialize with signature to verify };
ret = wc_DsaVerify(hash, signature, &key, &verified);
if (ret != 0) {
    // error processing verify request
} else if (answer == 0) {
    // invalid signature
}
```

```
int wc_DsaCheckPubKey(
    DsaKey * key
)
```

This function validates DSA domain parameters and the public key contained in the given DsaKey. It performs the following checks (a subset of the requirements in FIPS 186-4 and NIST SP 800-89): p > 1 and q > 1; q divides (p - 1); 1 < g < p; 1 < y < p; g^q mod p == 1 (so the multiplicative order of g divides q); and y^q mod p == 1 (so the multiplicative order of y divides q). The verify routines (wc_DsaVerify, wc_DsaVerify_ex) call this internally before any signature math runs, but the function is also exposed so callers can validate keys at import or decode time.

Parameters:

- **key** pointer to a DsaKey structure populated with p, q, g, and y

See:

- wc_DsaVerify
- wc_DsaPublicKeyDecode
- wc_DsaImportParamsRaw
- wc_DsaImportParamsRawCheck

Return:

- 0 Returned when the key and domain parameters pass validation.
- BAD_FUNC_ARG Returned when key is NULL, or when the key fails any of the validation checks listed above.
- MEMORY_E Returned on memory allocation failure (small-stack builds only).
- MP_INIT_E Returned when mp_int initialization fails.

- Other negative MP error codes (e.g. MP_EXPTMOD_E) Returned on internal big-integer failures.

Note: this function does not run primality tests on p or q. Full FIPS 186-4 domain-parameter validation additionally requires that p and q be prime; callers that need that level of assurance should use wc_DsaImportParamsRawCheck() (which validates p) and/or run mp_prime_is_prime_ex() on q at import time.

Example

```
DsaKey key;
int ret;
wc_InitDsaKey(&key);
// ... import or decode the public key into &key ...
ret = wc_DsaCheckPubKey(&key);
if (ret != 0) {
    // domain parameters or public key are invalid; reject
}
wc_FreeDsaKey(&key);
```

```
int wc_DsaPublicKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    DsaKey * key,
    word32 inSz
)
```

This function decodes a DER formatted certificate buffer containing a DSA public key, and stores the key in the given DsaKey structure. It also sets the inOutIdx parameter according to the length of the input read.

Parameters:

- **input** pointer to the buffer containing the DER formatted DSA public key
- **inOutIdx** pointer to an integer in which to store the final index of the certificate read
- **key** pointer to the DsaKey structure in which to store the public key
- **inSz** size of the input buffer

See:

- wc_InitDsaKey
- wc_DsaPrivateKeyDecode

Return:

- 0 Returned on successfully setting the public key for the DsaKey object
- ASN_PARSE_E Returned if there is an error in the encoding while reading the certificate buffer
- ASN_DH_KEY_E Returned if one of the DSA parameters is incorrectly formatted

Example

```
int ret, idx=0;

DsaKey key;
wc_InitDsaKey(&key);
byte derBuff[] = { // DSA public key};
ret = wc_DsaPublicKeyDecode(derBuff, &idx, &key, inSz);
if (ret != 0) {
    // error reading public key
}
```

```
int wc_DsaPrivateKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    DsaKey * key,
    word32 inSz
)
```

This function decodes a DER formatted certificate buffer containing a DSA private key, and stores the key in the given DsaKey structure. It also sets the inOutIdx parameter according to the length of the input read.

Parameters:

- **input** pointer to the buffer containing the DER formatted DSA private key
- **inOutIdx** pointer to an integer in which to store the final index of the certificate read
- **key** pointer to the DsaKey structure in which to store the private key
- **inSz** size of the input buffer

See:

- wc_InitDsaKey
- wc_DsaPublicKeyDecode

Return:

- 0 Returned on successfully setting the private key for the DsaKey object
- ASN_PARSE_E Returned if there is an error in the encoding while reading the certificate buffer
- ASN_DH_KEY_E Returned if one of the DSA parameters is incorrectly formatted

Example

```
int ret, idx=0;

DsaKey key;
wc_InitDsaKey(&key);
byte derBuff[] = { // DSA private key };
ret = wc_DsaPrivateKeyDecode(derBuff, &idx, &key, inSz);
if (ret != 0) {
    // error reading private key
}
```

```
int wc_DsaKeyToDer(
    DsaKey * key,
    byte * output,
    word32 outLen
)
```

Convert DsaKey key to DER format, write to output (outLen), return bytes written.

Parameters:

- **key** Pointer to DsaKey structure to convert.
- **output** Pointer to output buffer for converted key.
- **outLen** Length of output buffer.

See:

- wc_InitDsaKey
- wc_FreeDsaKey
- wc_MakeDsaKey

Return:

- outLen Success, number of bytes written
- BAD_FUNC_ARG key or output are null or key->type is not DSA_PRIVATE.
- MEMORY_E Error allocating memory.

Example

```
DsaKey key;
WC_RNG rng;
int derSz;
int bufferSize = // Sufficient buffer size;
byte der[bufferSize];

wc_InitDsaKey(&key);
wc_InitRng(&rng);
wc_MakeDsaKey(&rng, &key);
derSz = wc_DsaKeyToDer(&key, der, bufferSize);
```

```
int wc_MakeDsaKey(
    WC_RNG * rng,
    DsaKey * dsa
)
```

Create a DSA key.

Parameters:

- **rng** Pointer to WC_RNG structure.
- **dsa** Pointer to DsaKey structure.

See:

- wc_InitDsaKey
- wc_FreeDsaKey
- wc_DsaSign

Return:

- MP_OKAY Success
- BAD_FUNC_ARG Either rng or dsa is null.
- MEMORY_E Couldn't allocate memory for buffer.
- MP_INIT_E Error initializing mp_int

Example

```
WC_RNG rng;
DsaKey dsa;
wc_InitRng(&rng);
wc_InitDsa(&dsa);
if(wc_MakeDsaKey(&rng, &dsa) != 0)
{
    // Error creating key
}
```

```
int wc_MakeDsaParameters(
    WC_RNG * rng,
    int modulus_size,
    DsaKey * dsa
)
```

FIPS 186-4 defines valid for modulus_size values as (1024, 160) (2048, 256) (3072, 256)

Parameters:

- **rng** pointer to wolfCrypt rng.
- **modulus_size** 1024, 2048, or 3072 are valid values.
- **dsa** Pointer to a DsaKey structure.

See:

- wc_MakeDsaKey
- wc_DsaKeyToDer
- wc_InitDsaKey

Return:

- 0 Success
- BAD_FUNC_ARG rng or dsa is null or modulus_size is invalid.
- MEMORY_E Error attempting to allocate memory.

Example

```
DsaKey key;
WC_RNG rng;
wc_InitDsaKey(&key);
wc_InitRng(&rng);
if(wc_MakeDsaParameters(&rng, 1024, &genKey) != 0)
{
    // Handle error
}
```

```
int wc_InitDsaKey_h(
    DsaKey * key,
    void * h
)
```

Initializes DSA key with heap hint.

Parameters:

- **key** DSA key structure
- **h** Heap hint for memory allocation

See: wc_InitDsaKey

Return:

- 0 on success
- negative on failure

Example

```
DsaKey key;
int ret = wc_InitDsaKey_h(&key, NULL);
```

```
int wc_DsaSign_ex(
    const byte * digest,
    word32 digestSz,
    byte * out,
    DsaKey * key,
    WC_RNG * rng
)
```

Signs digest with extended parameters.

Parameters:

- **digest** Digest to sign
- **digestSz** Digest size
- **out** Output signature buffer
- **key** DSA key
- **rng** Random number generator

See: wc_DsaSign

Return:

- 0 on success
- negative on failure

Example

```
byte digest[WC_SHA_DIGEST_SIZE];
byte sig[40];
WC_RNG rng;
int ret = wc_DsaSign_ex(digest, sizeof(digest), sig, &key,
                        &rng);
```

```
int wc_DsaVerify_ex(
    const byte * digest,
    word32 digestSz,
    const byte * sig,
    DsaKey * key,
    int * answer
)
```

Verifies signature with extended parameters.

Parameters:

- **digest** Digest
- **digestSz** Digest size
- **sig** Signature buffer
- **key** DSA key
- **answer** Verification result

See: wc_DsaVerify

Return:

- 0 on success
- negative on failure

Example

```
byte digest[WC_SHA_DIGEST_SIZE];
byte sig[40];
int answer;
int ret = wc_DsaVerify_ex(digest, sizeof(digest), sig, &key,
                          &answer);
```

```
int wc_SetDsaPublicKey(
    byte * output,
    DsaKey * key,
    int outLen,
    int with_header
)
```

Sets DSA public key in output buffer.

Parameters:

- **output** Output buffer
- **key** DSA key
- **outLen** Output buffer length
- **with_header** Include header flag

See: wc_DsaKeyToPublicDer

Return:

- Size on success
- negative on failure

Example

```
byte output[256];
int ret = wc_SetDsaPublicKey(output, &key, sizeof(output), 1);
```

```
int wc_DsaKeyToPublicDer(
    DsaKey * key,
    byte * output,
    word32 outLen
)
```

Converts DSA key to public DER format.

Parameters:

- **key** DSA key
- **output** Output buffer
- **outLen** Output buffer length

See: wc_SetDsaPublicKey

Return:

- Size on success
- negative on failure

Example

```
DsaKey key;
WC_RNG rng;
byte output[256];

// Initialize key and RNG
wc_InitDsaKey(&key);
wc_InitRng(&rng);

// Generate DSA key or import existing key
wc_MakeDsaKey(&rng, &key);

// Convert to public DER format
int ret = wc_DsaKeyToPublicDer(&key, output, sizeof(output));
if (ret > 0) {
    // output contains DER encoded public key of size ret
}

wc_FreeDsaKey(&key);
wc_FreeRng(&rng);
```

```
int wc_DsaImportParamsRaw(
    DsaKey * dsa,
    const char * p,
    const char * q,
    const char * g
)
```

Imports DSA parameters from raw format. The parameters p, q, and g must be provided as ASCII hexadecimal strings (without 0x prefix). These represent the DSA domain parameters: p is the prime modulus, q is the prime divisor (subgroup order), and g is the generator.

Parameters:

- **dsa** DSA key structure (must be initialized)
- **p** P parameter as ASCII hex string (prime modulus)
- **q** Q parameter as ASCII hex string (prime divisor/subgroup order)
- **g** G parameter as ASCII hex string (generator)

See:

- wc_DsaImportParamsRawCheck

- wc_InitDsaKey

Return:

- 0 on success
- negative on failure

Example

```
DsaKey dsa;
wc_InitDsaKey(&dsa);

// DSA parameters as ASCII hexadecimal strings (example values)
const char* pStr = "E0A67598CD1B763BC98C8ABB333E5DDA0CD3AA0E5E1F"
                   "B5BA8A7B4EABC10BA338FAE06DD4B90FDA70D7CF0CB0"
                   "C638BE3341BEC0AF8A7330A3307DED2299A0EE606DF0"
                   "35177A239C34A912C202AA5F83B9C4A7CF0235B5316B"
                   "FC6EFB9A248411258B30B839AF172440F32563056CB6"
                   "7A861158DDD90E6A894C72A5BBEF9E286C6B";
const char* qStr = "E950511EAB424B9A19A2AEB4E159B7844C589C4F";
const char* gStr = "D29D5121B0423C2769AB21843E5A3240FF19CACC792D"
                   "C6E7925E6D1A4E6E4E3D119A3D133C8D3C8C8C8C8C8C"
                   "8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C"
                   "8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C";

int ret = wc_DsaImportParamsRaw(&dsa, pStr, qStr, gStr);
if (ret == 0) {
    // DSA parameters successfully imported
    // Can now use dsa for key generation or signing
}
wc_FreeDsaKey(&dsa);
```

```
int wc_DsaImportParamsRawCheck(
    DsaKey * dsa,
    const char * p,
    const char * q,
    const char * g,
    int trusted,
    WC_RNG * rng
)
```

Imports DSA parameters from raw format with optional validation. The parameters p, q, and g must be provided as ASCII hexadecimal strings (without 0x prefix). The trusted parameter controls whether the prime p is validated: when trusted=1, prime checking is skipped (use when parameters come from a trusted source); when trusted=0, performs full primality testing on p (recommended for untrusted sources).

Parameters:

- **dsa** DSA key structure (must be initialized)
- **p** P parameter as ASCII hex string (prime modulus)

- **q** Q parameter as ASCII hex string (prime divisor/subgroup order)
- **g** G parameter as ASCII hex string (generator)
- **trusted** If 1, skip prime validation (trusted source); if 0, perform full primality test on p
- **rng** Random number generator (required when trusted=0 for primality testing)

See:

- wc_DsaImportParamsRaw
- wc_InitDsaKey

Return:

- 0 on success
- DH_CHECK_PUB_E if p fails primality test (when trusted=0)
- negative on other failures

Example

```
DsaKey dsa;
WC_RNG rng;

// Initialize DSA key and RNG
wc_InitDsaKey(&dsa);
wc_InitRng(&rng);

// DSA parameters as ASCII hexadecimal strings
const char* pStr = "E0A67598CD1B763BC98C8ABB333E5DDA0CD3AA0E5E1F"
                   "B5BA8A7B4EABC10BA338FAE06DD4B90FDA70D7CF0CB0"
                   "C638BE3341BEC0AF8A7330A3307DED2299A0EE606DF0"
                   "35177A239C34A912C202AA5F83B9C4A7CF0235B5316B"
                   "FC6EFB9A248411258B30B839AF172440F32563056CB6"
                   "7A861158DDD90E6A894C72A5BBEF9E286C6B";
const char* qStr = "E950511EAB424B9A19A2AEB4E159B7844C589C4F";
const char* gStr = "D29D5121B0423C2769AB21843E5A3240FF19CACC792D"
                   "C6E7925E6D1A4E6E4E3D119A3D133C8D3C8C8C8C8C8C"
                   "8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C"
                   "8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C";

// Import with validation (trusted=0 performs primality test on p)
int ret = wc_DsaImportParamsRawCheck(&dsa, pStr, qStr, gStr, 0,
                                     &rng);
if (ret == 0) {
    // Parameters imported and validated successfully
}

wc_FreeDsaKey(&dsa);
wc_FreeRng(&rng);
```

```
int wc_DsaExportParamsRaw(
    DsaKey * dsa,
    byte * p,
```

```
    word32 * pSz,
    byte * q,
    word32 * qSz,
    byte * g,
    word32 * gSz
)
```

Exports DSA parameters to raw format.

Parameters:

- **dsa** DSA key structure
- **p** P parameter buffer
- **pSz** P parameter size (in/out)
- **q** Q parameter buffer
- **qSz** Q parameter size (in/out)
- **g** G parameter buffer
- **gSz** G parameter size (in/out)

See: wc_DsaImportParamsRaw

Return:

- 0 on success
- negative on failure

Example

```
byte p[256], q[32], g[256];
word32 pSz = sizeof(p), qSz = sizeof(q), gSz = sizeof(g);
int ret = wc_DsaExportParamsRaw(&dsa, p, &pSz, q, &qSz, g,
                                &gSz);
```

```
int wc_DsaExportKeyRaw(
    DsaKey * dsa,
    byte * x,
    word32 * xSz,
    byte * y,
    word32 * ySz
)
```

Exports DSA key to raw format.

Parameters:

- **dsa** DSA key structure
- **x** Private key buffer
- **xSz** Private key size (in/out)
- **y** Public key buffer
- **ySz** Public key size (in/out)

See: wc_DsaImportParamsRaw

Return:

- 0 on success
- negative on failure

Example

```
byte x[32], y[256];
word32 xSz = sizeof(x), ySz = sizeof(y);
int ret = wc_DsaExportKeyRaw(&dsa, x, &xSz, y, &ySz);
```

B.23 Algorithms - Diffie-Hellman

B.22.2.22 function wc_DsaExportKeyRaw

B.23.1 Functions

	Name
int	**wc_InitDhKey**(DhKey * key)This function initializes a Diffie-Hellman key for use in negotiating a secure secret key with the Diffie-Hellman exchange protocol.
int	**wc_FreeDhKey**(DhKey * key)This function frees a Diffie-Hellman key after it has been used to negotiate a secure secret key with the Diffie-Hellman exchange protocol.
int	**wc_DhGenerateKeyPair**(DhKey * key, WC_RNG * rng, byte * priv, word32 * privSz, byte * pub, word32 * pubSz)This function generates a public/private key pair based on the Diffie-Hellman public parameters, storing the private key in priv and the public key in pub. It takes an initialized Diffie-Hellman key and an initialized rng structure.
int	**wc_DhAgree**(DhKey * key, byte * agree, word32 * agreeSz, const byte * priv, word32 privSz, const byte * otherPub, word32 pubSz)This function generates an agreed upon secret key based on a local private key and a received public key. If completed on both sides of an exchange, this function generates an agreed upon secret key for symmetric communication. On successfully generating a shared secret key, the size of the secret key written will be stored in agreeSz.

	Name
int	**wc_DhKeyDecode**(const byte * input, word32 * inOutIdx, DhKey * key, word32 inSz)This function decodes a Diffie-Hellman key from the given input buffer containing the key in DER format. It stores the result in the DhKey structure.
int	**wc_DhSetKey**(DhKey * key, const byte * p, word32 pSz, const byte * g, word32 gSz)This function sets the key for a DhKey structure using the input private key parameters. Unlike wc_DhKeyDecode, this function does not require that the input key be formatted in DER format, and instead simply accepts the parsed input parameters p (prime) and g (base).
int	**wc_DhParamsLoad**(const byte * input, word32 inSz, byte * p, word32 * pInOutSz, byte * g, word32 * gInOutSz)This function loads the Diffie_Hellman parameters, p (prime) and g (base) out of the given input buffer, DER formatted.
int	**wolfSSL_i2d_DHparams**(const WOLFSSL_DH * dh, unsigned char ** out)Encodes DH parameters to DER format for OpenSSL compatibility.
WOLFSSL_DH *	**wolfSSL_DH_new**(void)Allocates and initializes a new DH structure for OpenSSL compatibility.
WOLFSSL_DH *	**wolfSSL_DH_new_by_nid**(int nid)Creates a new DH structure with named group parameters.
void	**wolfSSL_DH_free**(WOLFSSL_DH * dh)Frees a DH structure.
WOLFSSL_DH *	**wolfSSL_DH_dup**(WOLFSSL_DH * dh)Duplicates a DH structure.
int	**wolfSSL_DH_up_ref**(WOLFSSL_DH * dh)Increments reference count for DH structure.
int	**wolfSSL_DH_check**(const WOLFSSL_DH * dh, int * codes)Validates DH parameters.
int	**wolfSSL_DH_size**(WOLFSSL_DH * dh)Returns size of DH key in bytes.
int	**wolfSSL_DH_generate_key**(WOLFSSL_DH * dh)Generates DH public/private key pair.
int	**wolfSSL_DH_compute_key**(unsigned char * key, const WOLFSSL_BIGNUM * pub, WOLFSSL_DH * dh)Computes shared secret from peer's public key.
int	**wolfSSL_DH_compute_key_padded**(unsigned char * key, const WOLFSSL_BIGNUM * otherPub, WOLFSSL_DH * dh)Computes shared secret with zero-padding to DH size.
int	**wolfSSL_DH_LoadDer**(WOLFSSL_DH * dh, const unsigned char * derBuf, int derSz)Loads DH parameters from DER buffer.

	Name
int	**wolfSSL_DH_set_length**(WOLFSSL_DH * dh, long len)Sets optional private key length.
int	**wolfSSL_DH_set0_pqg**(WOLFSSL_DH * dh, WOLFSSL_BIGNUM * p, WOLFSSL_BIGNUM * q, WOLFSSL_BIGNUM * g)Sets DH parameters p, q, and g.
WOLFSSL_DH *	**wolfSSL_DH_get_2048_256**(void)Returns DH parameters for 2048-bit MODP group with 256-bit subgroup.
const DhParams *	**wc_Dh_ffdhe2048_Get**(void)Returns FFDHE 2048-bit group parameters.
const DhParams *	**wc_Dh_ffdhe3072_Get**(void)Returns FFDHE 3072-bit group parameters.
const DhParams *	**wc_Dh_ffdhe4096_Get**(void)Returns FFDHE 4096-bit group parameters.
const DhParams *	**wc_Dh_ffdhe6144_Get**(void)Returns FFDHE 6144-bit group parameters.
const DhParams *	**wc_Dh_ffdhe8192_Get**(void)Returns FFDHE 8192-bit group parameters.
int	**wc_InitDhKey_ex**(DhKey * key, void * heap, int devId)Initializes DH key with heap hint and device ID.
int	**wc_DhAgree_ct**(DhKey * key, byte * agree, word32 * agreeSz, const byte * priv, word32 privSz, const byte * otherPub, word32 pubSz)Computes shared secret with constant-time operations.
int	**wc_DhSetNamedKey**(DhKey * key, int name)Sets DH key to use named group parameters.
int	**wc_DhGetNamedKeyParamSize**(int name, word32 * p, word32 * g, word32 * q)Gets parameter sizes for named group.
word32	**wc_DhGetNamedKeyMinSize**(int name)Gets minimum key size for named group.
int	**wc_DhCmpNamedKey**(int name, int noQ, const byte * p, word32 pSz, const byte * g, word32 gSz, const byte * q, word32 qSz)Compares parameters against named group.
int	**wc_DhCopyNamedKey**(int name, byte * p, word32 * pSz, byte * g, word32 * gSz, byte * q, word32 * qSz)Copies named group parameters to buffers.
int	**wc_DhGeneratePublic**(DhKey * key, byte * priv, word32 privSz, byte * pub, word32 * pubSz)Generates public key from private key.
int	**wc_DhImportKeyPair**(DhKey * key, const byte * priv, word32 privSz, const byte * pub, word32 pubSz)Imports private and/or public key into DH key.

	Name
int	**wc_DhExportKeyPair**(DhKey * key, byte * priv, word32 * pPrivSz, byte * pub, word32 * pPubSz)Exports private and public key from DH key.
int	**wc_DhCheckPubValue**(const byte * prime, word32 primeSz, const byte * pub, word32 pubSz)Validates public key value.
int	**wc_DhCheckKeyPair**(DhKey * key, const byte * pub, word32 pubSz, const byte * priv, word32 privSz)Checks DH keys for pair_wise consistency per process in SP 800_56Ar3, section 5.6.2.1.4, method (b) for FFC.
int	**wc_DhCheckPrivKey**(DhKey * key, const byte * priv, word32 pubSz)Check DH private key for invalid numbers.
int	**wc_DhCheckPrivKey_ex**(DhKey * key, const byte * priv, word32 pubSz, const byte * prime, word32 primeSz)
int	**wc_DhCheckPubKey**(DhKey * key, const byte * pub, word32 pubSz)
int	**wc_DhCheckPubKey_ex**(DhKey * key, const byte * pub, word32 pubSz, const byte * prime, word32 primeSz)
int	**wc_DhExportParamsRaw**(DhKey * dh, byte * p, word32 * pSz, byte * q, word32 * qSz, byte * g, word32 * gSz)
int	**wc_DhGenerateParams**(WC_RNG * rng, int modSz, DhKey * dh)
int	**wc_DhSetCheckKey**(DhKey * key, const byte * p, word32 pSz, const byte * g, word32 gSz, const byte * q, word32 qSz, int trusted, WC_RNG * rng)
int	**wc_DhSetKey_ex**(DhKey * key, const byte * p, word32 pSz, const byte * g, word32 gSz, const byte * q, word32 qSz)

B.23.2 Functions Documentation

```
int wc_InitDhKey(
    DhKey * key
)
```

This function initializes a Diffie-Hellman key for use in negotiating a secure secret key with the Diffie-Hellman exchange protocol.

Parameters:

- **key** pointer to the DhKey structure to initialize for use with secure key exchanges

See:

- wc_FreeDhKey
- wc_DhGenerateKeyPair

Return: none No returns.

Example

```
DhKey key;
wc_InitDhKey(&key); // initialize DH key
```

```
int wc_FreeDhKey(
    DhKey * key
)
```

This function frees a Diffie-Hellman key after it has been used to negotiate a secure secret key with the Diffie-Hellman exchange protocol.

Parameters:

- **key** pointer to the DhKey structure to free

See: wc_InitDhKey

Return: none No returns.

Example

```
DhKey key;
// initialize key, perform key exchange

wc_FreeDhKey(&key); // free DH key to avoid memory leaks
```

```
int wc_DhGenerateKeyPair(
    DhKey * key,
    WC_RNG * rng,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz
)
```

This function generates a public/private key pair based on the Diffie-Hellman public parameters, storing the private key in priv and the public key in pub. It takes an initialized Diffie-Hellman key and an initialized rng structure.

Parameters:

- **key** pointer to the DhKey structure from which to generate the key pair
- **rng** pointer to an initialized random number generator (rng) with which to generate the keys
- **priv** pointer to a buffer in which to store the private key

- **privSz** will store the size of the private key written to priv
- **pub** pointer to a buffer in which to store the public key
- **pubSz** will store the size of the private key written to pub

See:

- wc_InitDhKey
- wc_DhSetKey
- wc_DhKeyDecode

Return:

- BAD_FUNC_ARG Returned if there is an error parsing one of the inputs to this function
- RNG_FAILURE_E Returned if there is an error generating a random number using rng
- MP_INIT_E May be returned if there is an error in the math library while generating the public key
- MP_READ_E May be returned if there is an error in the math library while generating the public key
- MP_EXPTMOD_E May be returned if there is an error in the math library while generating the public key
- MP_TO_E May be returned if there is an error in the math library while generating the public key

Example

```
DhKey key;
int ret;
byte priv[256];
byte pub[256];
word32 privSz, pubSz;

wc_InitDhKey(&key); // initialize key
// Set DH parameters using wc_DhSetKey or wc_DhKeyDecode
WC_RNG rng;
wc_InitRng(&rng); // initialize rng
ret = wc_DhGenerateKeyPair(&key, &rng, priv, &privSz, pub, &pubSz);
```

```
int wc_DhAgree(
    DhKey * key,
    byte * agree,
    word32 * agreeSz,
    const byte * priv,
    word32 privSz,
    const byte * otherPub,
    word32 pubSz
)
```

This function generates an agreed upon secret key based on a local private key and a received public key. If completed on both sides of an exchange, this function generates an agreed upon secret key for symmetric communication. On successfully generating a shared secret key, the size of the secret key written will be stored in agreeSz.

Parameters:

- **key** pointer to the DhKey structure to use to compute the shared key
- **agree** pointer to the buffer in which to store the secret key
- **agreeSz** will hold the size of the secret key after successful generation
- **priv** pointer to the buffer containing the local secret key
- **privSz** size of the local secret key
- **otherPub** pointer to a buffer containing the received public key
- **pubSz** size of the received public key

See: wc_DhGenerateKeyPair

Return:

- 0 Returned on successfully generating an agreed upon secret key
- MP_INIT_E May be returned if there is an error while generating the shared secret key
- MP_READ_E May be returned if there is an error while generating the shared secret key
- MP_EXPTMOD_E May be returned if there is an error while generating the shared secret key
- MP_TO_E May be returned if there is an error while generating the shared secret key

Example

```
DhKey key;
int ret;
byte priv[256];
byte agree[256];
word32 agreeSz;

// initialize key, set key prime and base
// wc_DhGenerateKeyPair -- store private key in priv
byte pub[] = { // initialized with the received public key };
ret = wc_DhAgree(&key, agree, &agreeSz, priv, sizeof(priv), pub,
sizeof(pub));
if ( ret != 0 ) {
    // error generating shared key
}
```

```
int wc_DhKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    DhKey * key,
    word32 inSz
)
```

This function decodes a Diffie-Hellman key from the given input buffer containing the key in DER format. It stores the result in the DhKey structure.

Parameters:

- **input** pointer to the buffer containing the DER formatted Diffie-Hellman key

- **inOutIdx** pointer to an integer in which to store the index parsed to while decoding the key
- **key** pointer to the DhKey structure to initialize with the input key
- **inSz** length of the input buffer. Gives the max length that may be read

See: wc_DhSetKey

Return:

- 0 Returned on successfully decoding the input key
- ASN_PARSE_E Returned if there is an error parsing the sequence of the input
- ASN_DH_KEY_E Returned if there is an error reading the private key parameters from the parsed input

Example

```
DhKey key;
word32 idx = 0;

byte keyBuff[1024];
// initialize with DER formatted key
wc_DhKeyInit(&key);
ret = wc_DhKeyDecode(keyBuff, &idx, &key, sizeof(keyBuff));

if ( ret != 0 ) {
    // error decoding key
}
```

```
int wc_DhSetKey(
    DhKey * key,
    const byte * p,
    word32 pSz,
    const byte * g,
    word32 gSz
)
```

This function sets the key for a DhKey structure using the input private key parameters. Unlike wc_DhKeyDecode, this function does not require that the input key be formatted in DER format, and instead simply accepts the parsed input parameters p (prime) and g (base).

Parameters:

- **key** pointer to the DhKey structure on which to set the key
- **p** pointer to the buffer containing the prime for use with the key
- **pSz** length of the input prime
- **g** pointer to the buffer containing the base for use with the key
- **gSz** length of the input base

See: wc_DhKeyDecode

Return:

- 0 Returned on successfully setting the key
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL
- MP_INIT_E Returned if there is an error initializing the key parameters for storage
- ASN_DH_KEY_E Returned if there is an error reading in the DH key parameters p and g

Example

```
DhKey key;

byte p[] = { // initialize with prime };
byte g[] = { // initialize with base };
wc_DhKeyInit(&key);
ret = wc_DhSetKey(key, p, sizeof(p), g, sizeof(g));

if ( ret != 0 ) {
    // error setting key
}
```

```
int wc_DhParamsLoad(
    const byte * input,
    word32 inSz,
    byte * p,
    word32 * pInOutSz,
    byte * g,
    word32 * gInOutSz
)
```

This function loads the Diffie-Hellman parameters, p (prime) and g (base) out of the given input buffer, DER formatted.

Parameters:

- **input** pointer to a buffer containing a DER formatted Diffie-Hellman certificate to parse
- **inSz** size of the input buffer
- **p** pointer to a buffer in which to store the parsed prime
- **pInOutSz** pointer to a word32 object containing the available size in the p buffer. Will be overwritten with the number of bytes written to the buffer after completing the function call
- **g** pointer to a buffer in which to store the parsed base
- **gInOutSz** pointer to a word32 object containing the available size in the g buffer. Will be overwritten with the number of bytes written to the buffer after completing the function call

See:

- wc_DhSetKey
- wc_DhKeyDecode

Return:

- 0 Returned on successfully extracting the DH parameters
- ASN_PARSE_E Returned if an error occurs while parsing the DER formatted DH certificate

- BUFFER_E Returned if there is inadequate space in p or g to store the parsed parameters

Example

```
byte dhCert[] = { initialize with DER formatted certificate };
byte p[MAX_DH_SIZE];
byte g[MAX_DH_SIZE];
word32 pSz = MAX_DH_SIZE;
word32 gSz = MAX_DH_SIZE;

ret = wc_DhParamsLoad(dhCert, sizeof(dhCert), p, &pSz, g, &gSz);
if ( ret != 0 ) {
    // error parsing inputs
}
```

```
int wolfSSL_i2d_DHparams(
    const WOLFSSL_DH * dh,
    unsigned char ** out
)
```

Encodes DH parameters to DER format for OpenSSL compatibility.

Parameters:

- **dh** DH parameters to encode
- **out** Output buffer pointer (if *out is NULL, allocates buffer)

See: wolfSSL_DH_new

Return:

- Length of DER encoding on success
- Negative on error

Example

```
WOLFSSL_DH* dh = wolfSSL_DH_new();
unsigned char* der = NULL;
int derSz = wolfSSL_i2d_DHparams(dh, &der);
if (derSz > 0) {
    // use der buffer
    XFREE(der, NULL, DYNAMIC_TYPE_OPENSSL);
}
```

```
WOLFSSL_DH * wolfSSL_DH_new(
    void
)
```

Allocates and initializes a new DH structure for OpenSSL compatibility.

See:

- wolfSSL_DH_free
- wolfSSL_DH_generate_key

Return:

- Pointer to WOLFSSL_DH on success
- NULL on failure

Example

```
WOLFSSL_DH* dh = wolfSSL_DH_new();
if (dh == NULL) {
    // error allocating DH
}
// use dh
wolfSSL_DH_free(dh);
```

```
WOLFSSL_DH * wolfSSL_DH_new_by_nid(
    int nid
)
```

Creates a new DH structure with named group parameters.

Parameters:

- **nid** Named group identifier (e.g., NID_ffdhe2048)

See: wolfSSL_DH_new

Return:

- Pointer to WOLFSSL_DH on success
- NULL on failure

Example

```
WOLFSSL_DH* dh = wolfSSL_DH_new_by_nid(NID_ffdhe2048);
if (dh == NULL) {
    // error creating DH with named group
}
```

```
void wolfSSL_DH_free(
    WOLFSSL_DH * dh
)
```

Frees a DH structure.

Parameters:

- **dh** DH structure to free

See: wolfSSL_DH_new

Example

```
WOLFSSL_DH* dh = wolfSSL_DH_new();
// use dh
wolfSSL_DH_free(dh);
```

```
WOLFSSL_DH * wolfSSL_DH_dup(
    WOLFSSL_DH * dh
)
```

Duplicates a DH structure.

Parameters:

- **dh** DH structure to duplicate

See: wolfSSL_DH_new

Return:

- Pointer to new WOLFSSL_DH on success
- NULL on failure

Example

```
WOLFSSL_DH* dh = wolfSSL_DH_new();
WOLFSSL_DH* dhCopy = wolfSSL_DH_dup(dh);
```

```
int wolfSSL_DH_up_ref(
    WOLFSSL_DH * dh
)
```

Increments reference count for DH structure.

Parameters:

- **dh** DH structure to increment reference

See: wolfSSL_DH_free

Return:

- 1 on success
- 0 on failure

Example

```
WOLFSSL_DH* dh = wolfSSL_DH_new();
int ret = wolfSSL_DH_up_ref(dh);
```

```
int wolfSSL_DH_check(
    const WOLFSSL_DH * dh,
    int * codes
)
```

Validates DH parameters.

Parameters:

- **dh** DH parameters to check
- **codes** Output for validation error codes

See: wolfSSL_DH_generate_key

Return:

- 1 on success
- 0 on failure

Example

```
WOLFSSL_DH* dh = wolfSSL_DH_new();
int codes;
int ret = wolfSSL_DH_check(dh, &codes);
if (ret != 1 || codes != 0) {
    // validation failed
}
```

```
int wolfSSL_DH_size(
    WOLFSSL_DH * dh
)
```

Returns size of DH key in bytes.

Parameters:

- **dh** DH structure

See: wolfSSL_DH_new

Return:

- Key size in bytes on success
- -1 on failure

Example

```
WOLFSSL_DH* dh = wolfSSL_DH_new();
int size = wolfSSL_DH_size(dh);
```

```
int wolfSSL_DH_generate_key(
    WOLFSSL_DH * dh
)
```

Generates DH public/private key pair.

Parameters:

- **dh** DH structure with parameters set

See: wolfSSL_DH_compute_key

Return:

- 1 on success
- 0 on failure

Example

```
WOLFSSL_DH* dh = wolfSSL_DH_new();
// set p and g parameters
int ret = wolfSSL_DH_generate_key(dh);
if (ret != 1) {
    // key generation failed
}
```

```
int wolfSSL_DH_compute_key(
    unsigned char * key,
    const WOLFSSL_BIGNUM * pub,
    WOLFSSL_DH * dh
)
```

Computes shared secret from peer's public key.

Parameters:

- **key** Output buffer for shared secret
- **pub** Peer's public key
- **dh** DH structure with private key

See: wolfSSL_DH_generate_key

Return:

- Length of shared secret on success
- -1 on failure

Example

```
WOLFSSL_DH* dh = wolfSSL_DH_new();
wolfSSL_DH_generate_key(dh);
byte secret[256];
WOLFSSL_BIGNUM* peerPub = NULL; // peer's public key
int secretSz = wolfSSL_DH_compute_key(secret, peerPub, dh);
```

```
int wolfSSL_DH_compute_key_padded(
    unsigned char * key,
    const WOLFSSL_BIGNUM * otherPub,
    WOLFSSL_DH * dh
)
```

Computes shared secret with zero-padding to DH size.

Parameters:

- **key** Output buffer for shared secret
- **otherPub** Peer's public key
- **dh** DH structure with private key

See: wolfSSL_DH_compute_key

Return:

- Length of shared secret on success
- -1 on failure

Example

```
WOLFSSL_DH* dh = wolfSSL_DH_new();
wolfSSL_DH_generate_key(dh);
byte secret[256];
```

```
WOLFSSL_BIGNUM* peerPub = NULL;
int secretSz = wolfSSL_DH_compute_key_padded(secret, peerPub, dh);
```

```
int wolfSSL_DH_LoadDer(
    WOLFSSL_DH * dh,
    const unsigned char * derBuf,
    int derSz
)
```

Loads DH parameters from DER buffer.

Parameters:

- **dh** DH structure to load into
- **derBuf** DER-encoded DH parameters
- **derSz** Size of DER buffer

See: wolfSSL_DH_new

Return:

- WOLFSSL_SUCCESS on success
- WOLFSSL_FAILURE on failure

Example

```
WOLFSSL_DH* dh = wolfSSL_DH_new();
byte derBuf[256];
int ret = wolfSSL_DH_LoadDer(dh, derBuf, sizeof(derBuf));
```

```
int wolfSSL_DH_set_length(
    WOLFSSL_DH * dh,
    long len
)
```

Sets optional private key length.

Parameters:

- **dh** DH structure
- **len** Private key length in bits

See: wolfSSL_DH_generate_key

Return:

- 1 on success
- 0 on failure

Example

```
WOLFSSL_DH* dh = wolfSSL_DH_new();
int ret = wolfSSL_DH_set_length(dh, 256);
```

```
int wolfSSL_DH_set0_pqg(
    WOLFSSL_DH * dh,
    WOLFSSL_BIGNUM * p,
    WOLFSSL_BIGNUM * q,
    WOLFSSL_BIGNUM * g
)
```

Sets DH parameters p, q, and g.

Parameters:

- **dh** DH structure
- **p** Prime modulus (takes ownership)
- **q** Subgroup order (takes ownership, can be NULL)
- **g** Generator (takes ownership)

See: wolfSSL_DH_generate_key

Return:

- 1 on success
- 0 on failure

Example

```
WOLFSSL_DH* dh = wolfSSL_DH_new();
WOLFSSL_BIGNUM *p = wolfSSL_BN_new();
WOLFSSL_BIGNUM *g = wolfSSL_BN_new();
// set p and g values
int ret = wolfSSL_DH_set0_pqg(dh, p, NULL, g);
```

```
WOLFSSL_DH * wolfSSL_DH_get_2048_256(
    void
)
```

Returns DH parameters for 2048-bit MODP group with 256-bit subgroup.

See: wolfSSL_DH_new_by_nid

Return:

- Pointer to WOLFSSL_DH on success
- NULL on failure

Example

```
WOLFSSL_DH* dh = wolfSSL_DH_get_2048_256();
if (dh == NULL) {
    // error getting standard group
}
```

```
const DhParams * wc_Dh_ffdhe2048_Get(
    void
)
```

Returns FFDHE 2048-bit group parameters.

See:

- wc_Dh_ffdhe3072_Get
- wc_Dh_ffdhe4096_Get
- wc_Dh_ffdhe6144_Get
- wc_Dh_ffdhe8192_Get

Return:

- Pointer to DhParams structure
- NULL if not compiled with HAVE_FFDHE_2048

Example

```
const DhParams* params = wc_Dh_ffdhe2048_Get();
if (params != NULL) {
    // use params
}
```

```
const DhParams * wc_Dh_ffdhe3072_Get(
    void
)
```

Returns FFDHE 3072-bit group parameters.

See:

- wc_Dh_ffdhe2048_Get
- wc_Dh_ffdhe4096_Get
- wc_Dh_ffdhe6144_Get
- wc_Dh_ffdhe8192_Get

Return:

- Pointer to DhParams structure
- NULL if not compiled with HAVE_FFDHE_3072

Example

```
const DhParams* params = wc_Dh_ffdhe3072_Get();
if (params != NULL) {
    // use params
}
```

```
const DhParams * wc_Dh_ffdhe4096_Get(
    void
)
```

Returns FFDHE 4096-bit group parameters.

See:

- wc_Dh_ffdhe2048_Get
- wc_Dh_ffdhe3072_Get
- wc_Dh_ffdhe6144_Get
- wc_Dh_ffdhe8192_Get

Return:

- Pointer to DhParams structure
- NULL if not compiled with HAVE_FFDHE_4096

Example

```
const DhParams* params = wc_Dh_ffdhe4096_Get();
if (params != NULL) {
    // use params
}
```

```
const DhParams * wc_Dh_ffdhe6144_Get(
    void
)
```

Returns FFDHE 6144-bit group parameters.

See:

- wc_Dh_ffdhe2048_Get
- wc_Dh_ffdhe3072_Get
- wc_Dh_ffdhe4096_Get
- wc_Dh_ffdhe8192_Get

Return:

- Pointer to DhParams structure
- NULL if not compiled with HAVE_FFDHE_6144

Example

```
const DhParams* params = wc_Dh_ffdhe6144_Get();
if (params != NULL) {
    // use params
}
```

```
const DhParams * wc_Dh_ffdhe8192_Get(
    void
)
```

Returns FFDHE 8192-bit group parameters.

See:

- wc_Dh_ffdhe2048_Get
- wc_Dh_ffdhe3072_Get
- wc_Dh_ffdhe4096_Get
- wc_Dh_ffdhe6144_Get

Return:

- Pointer to DhParams structure
- NULL if not compiled with HAVE_FFDHE_8192

Example

```
const DhParams* params = wc_Dh_ffdhe8192_Get();
if (params != NULL) {
    // use params
}
```

```
int wc_InitDhKey_ex(
    DhKey * key,
    void * heap,
    int devId
)
```

Initializes DH key with heap hint and device ID.

Parameters:

- **key** DH key to initialize
- **heap** Heap hint for memory allocation
- **devId** Device ID for hardware acceleration

See:

- wc_InitDhKey
- wc_FreeDhKey

Return:

- 0 on success
- BAD_FUNC_ARG if key is NULL

Example

```
DhKey key;
int ret = wc_InitDhKey_ex(&key, NULL, INVALID_DEVID);
if (ret != 0) {
    // error initializing key
}
```

```
int wc_DhAgree_ct(
    DhKey * key,
    byte * agree,
    word32 * agreeSz,
    const byte * priv,
    word32 privSz,
    const byte * otherPub,
    word32 pubSz
)
```

Computes shared secret with constant-time operations.

Parameters:

- **key** DH key with parameters
- **agree** Output buffer for shared secret
- **agreeSz** Input: buffer size, Output: secret size
- **priv** Private key
- **privSz** Private key size
- **otherPub** Peer's public key
- **pubSz** Peer's public key size

See: wc_DhAgree

Return:

- 0 on success
- BAD_FUNC_ARG if parameters are invalid

- BUFFER_E if output buffer too small

Example

```
DhKey key;
byte agree[256], priv[256], pub[256];
word32 agreeSz = sizeof(agree);
int ret = wc_DhAgree_ct(&key, agree, &agreeSz, priv,
                        sizeof(priv), pub, sizeof(pub));
```

```
int wc_DhSetNamedKey(
    DhKey * key,
    int name
)
```

Sets DH key to use named group parameters.

Parameters:

- **key** DH key to configure
- **name** Named group identifier

See: wc_DhGetNamedKeyParamSize

Return:

- 0 on success
- BAD_FUNC_ARG if parameters are invalid

Example

```
DhKey key;
wc_InitDhKey(&key);
int ret = wc_DhSetNamedKey(&key, WC_FFDHE_2048);
```

```
int wc_DhGetNamedKeyParamSize(
    int name,
    word32 * p,
    word32 * g,
    word32 * q
)
```

Gets parameter sizes for named group.

Parameters:

- **name** Named group identifier
- **p** Output for prime size

- **g** Output for generator size
- **q** Output for subgroup order size

See: wc_DhSetNamedKey

Return:

- 0 on success
- BAD_FUNC_ARG if parameters are invalid

Example

```
word32 pSz, gSz, qSz;
int ret = wc_DhGetNamedKeyParamSize(WC_FFDHE_2048, &pSz, &gSz,
                                    &qSz);
```

```
word32 wc_DhGetNamedKeyMinSize(
    int name
)
```

Gets minimum key size for named group.

Parameters:

- **name** Named group identifier

See: wc_DhSetNamedKey

Return:

- Minimum key size in bits
- 0 if invalid name

Example

```
word32 minSize = wc_DhGetNamedKeyMinSize(WC_FFDHE_2048);
```

```
int wc_DhCmpNamedKey(
    int name,
    int noQ,
    const byte * p,
    word32 pSz,
    const byte * g,
    word32 gSz,
    const byte * q,
    word32 qSz
)
```

Compares parameters against named group.

Parameters:

- **name** Named group identifier
- **noQ** 1 to skip q comparison
- **p** Prime modulus
- **pSz** Prime size
- **g** Generator
- **gSz** Generator size
- **q** Subgroup order
- **qSz** Subgroup order size

See: wc_DhSetNamedKey

Return:

- 0 if parameters match named group
- Non-zero if parameters don't match

Example

```
byte p[256], g[256];
int ret = wc_DhCmpNamedKey(WC_FFDHE_2048, 1, p, sizeof(p),
                           g, sizeof(g), NULL, 0);
```

```
int wc_DhCopyNamedKey(
    int name,
    byte * p,
    word32 * pSz,
    byte * g,
    word32 * gSz,
    byte * q,
    word32 * qSz
)
```

Copies named group parameters to buffers.

Parameters:

- **name** Named group identifier
- **p** Output buffer for prime
- **pSz** Input: buffer size, Output: prime size
- **g** Output buffer for generator
- **gSz** Input: buffer size, Output: generator size
- **q** Output buffer for subgroup order
- **qSz** Input: buffer size, Output: subgroup order size

See: wc_DhSetNamedKey

Return:

- 0 on success
- BAD_FUNC_ARG if parameters are invalid
- BUFFER_E if buffers too small

Example

```
byte p[512], g[512], q[512];
word32 pSz = sizeof(p), gSz = sizeof(g), qSz = sizeof(q);
int ret = wc_DhCopyNamedKey(WC_FFDHE_2048, p, &pSz, g, &gSz,
                            q, &qSz);
```

```
int wc_DhGeneratePublic(
    DhKey * key,
    byte * priv,
    word32 privSz,
    byte * pub,
    word32 * pubSz
)
```

Generates public key from private key.

Parameters:

- **key** DH key with parameters set
- **priv** Private key
- **privSz** Private key size
- **pub** Output buffer for public key
- **pubSz** Input: buffer size, Output: public key size

See: wc_DhGenerateKeyPair

Return:

- 0 on success
- BAD_FUNC_ARG if parameters are invalid

Example

```
DhKey key;
byte priv[256], pub[256];
word32 pubSz = sizeof(pub);
int ret = wc_DhGeneratePublic(&key, priv, sizeof(priv), pub,
                              &pubSz);
```

```
int wc_DhImportKeyPair(
    DhKey * key,
    const byte * priv,
```

```
    word32 privSz,
    const byte * pub,
    word32 pubSz
)
```

Imports private and/or public key into DH key.

Parameters:

- **key** DH key to import into
- **priv** Private key (can be NULL)
- **privSz** Private key size
- **pub** Public key (can be NULL)
- **pubSz** Public key size

See: wc_DhExportKeyPair

Return:

- 0 on success
- BAD_FUNC_ARG if parameters are invalid

Example

```
DhKey key;
byte priv[256], pub[256];
int ret = wc_DhImportKeyPair(&key, priv, sizeof(priv), pub,
                             sizeof(pub));
```

```
int wc_DhExportKeyPair(
    DhKey * key,
    byte * priv,
    word32 * pPrivSz,
    byte * pub,
    word32 * pPubSz
)
```

Exports private and public key from DH key.

Parameters:

- **key** DH key to export from
- **priv** Output buffer for private key
- **pPrivSz** Input: buffer size, Output: private key size
- **pub** Output buffer for public key
- **pPubSz** Input: buffer size, Output: public key size

See: wc_DhImportKeyPair

Return:

- 0 on success
- BAD_FUNC_ARG if parameters are invalid
- BUFFER_E if buffers too small

Example

```
DhKey key;
byte priv[256], pub[256];
word32 privSz = sizeof(priv), pubSz = sizeof(pub);
int ret = wc_DhExportKeyPair(&key, priv, &privSz, pub, &pubSz);
```

```
int wc_DhCheckPubValue(
    const byte * prime,
    word32 primeSz,
    const byte * pub,
    word32 pubSz
)
```

Validates public key value.

Parameters:

- **prime** Prime modulus
- **primeSz** Prime size
- **pub** Public key to validate
- **pubSz** Public key size

See: wc_DhCheckPubKey

Return:

- 0 if public key is valid
- BAD_FUNC_ARG if parameters are invalid
- MP_VAL if public key is invalid

Example

```
byte prime[256], pub[256];
int ret = wc_DhCheckPubValue(prime, sizeof(prime), pub,
                             sizeof(pub));
if (ret != 0) {
    // invalid public key
}
```

```
int wc_DhCheckKeyPair(
    DhKey * key,
    const byte * pub,
    word32 pubSz,
    const byte * priv,
```

```
    word32 privSz
)
```

Checks DH keys for pair-wise consistency per process in SP 800-56Ar3, section 5.6.2.1.4, method (b) for FFC.

```
int wc_DhCheckPrivKey(
    DhKey * key,
    const byte * priv,
    word32 pubSz
)
```

Check DH private key for invalid numbers.

```
int wc_DhCheckPrivKey_ex(
    DhKey * key,
    const byte * priv,
    word32 pubSz,
    const byte * prime,
    word32 primeSz
)
```

```
int wc_DhCheckPubKey(
    DhKey * key,
    const byte * pub,
    word32 pubSz
)
```

```
int wc_DhCheckPubKey_ex(
    DhKey * key,
    const byte * pub,
    word32 pubSz,
    const byte * prime,
    word32 primeSz
)
```

```
int wc_DhExportParamsRaw(
    DhKey * dh,
    byte * p,
    word32 * pSz,
    byte * q,
    word32 * qSz,
    byte * g,
```

```
    word32 * gSz
)



int wc_DhGenerateParams(
    WC_RNG * rng,
    int modSz,
    DhKey * dh
)



int wc_DhSetCheckKey(
    DhKey * key,
    const byte * p,
    word32 pSz,
    const byte * g,
    word32 gSz,
    const byte * q,
    word32 qSz,
    int trusted,
    WC_RNG * rng
)



int wc_DhSetKey_ex(
    DhKey * key,
    const byte * p,
    word32 pSz,
    const byte * g,
    word32 gSz,
    const byte * q,
    word32 qSz
)
```

Note: Parameters that are not one of the built-in FFDHE groups are untrusted, so the modulus is tested for primality with random Miller-Rabin witnesses. This function takes no RNG, so it instantiates a temporary one, which allocates memory and consumes entropy on every such call. When neither is available the test falls back to fixed witness bases, which a purpose-built composite can defeat. Call wc_DhSetCheckKey() with an initialized WC_RNG to supply your own.

B.24 Algorithms - ECC

B.23.2.47 function wc_DhSetKey_ex

B.24.1 Functions

	Name
int	**wc_EccPrivateKeyToDer**(ecc_key * key, byte * output, word32 outLen)Encodes ECC private key to DER format.
int	**wc_EccKeyDerSize**(ecc_key * key, int pub)Calculates DER encoded ECC key size.
int	**wc_EccPrivateKeyToPKCS8**(ecc_key * key, byte * output, word32 * outLen)Encodes ECC private key to PKCS#8 format.
int	**wc_EccKeyToPKCS8**(ecc_key * key, byte * output, word32 * outLen)Encodes ECC key pair to PKCS#8 format.
int	**wc_EccPublicKeyDerSize**(ecc_key * key, int with_AlgCurve)Calculates DER encoded ECC public key size.
int	**wc_ecc_make_key**(WC_RNG * rng, int keysize, ecc_key * key)This function generates a new ecc_key and stores it in key.
int	**wc_ecc_make_key_ex**(WC_RNG * rng, int keysize, ecc_key * key, int curve_id)This function generates a new ecc_key and stores it in key.
int	**wc_ecc_make_pub**(ecc_key * key, ecc_point * pubOut)wc_ecc_make_pub computes the public component from an ecc_key with an existing private component. If pubOut is supplied, the computed public key is stored there, else it is stored in the supplied ecc_key public component slot.
int	**wc_ecc_make_pub_ex**(ecc_key * key, ecc_point * pubOut, WC_RNG * rng)wc_ecc_make_pub_ex computes the public component from an ecc_key with an existing private component. If pubOut is supplied, the computed public key is stored there, else it is stored in the supplied ecc_key public component slot. The supplied rng, if non-NULL, is used to blind the private key value used in the computation.
int	**wc_ecc_check_key**(ecc_key * key)Perform sanity checks on ecc key validity.
void	**wc_ecc_key_free**(ecc_key * key)This function frees an ecc_key key after it has been used.
int	**wc_ecc_shared_secret**(ecc_key * private_key, ecc_key * public_key, byte * out, word32 * outlen)This function generates a new secret key using a local private key and a received public key. It stores this shared secret key in the buffer out and updates outlen to hold the number of bytes written to the output buffer.

	Name
int	**wc_ecc_shared_secret_ex**(ecc_key * private_key, ecc_point * point, byte * out, word32 * outlen)Create an ECC shared secret between private key and public point. The point is first given full public key validation on the private key's curve with wc_ecc_check_key() (SP 800_56Ar3 5.6.2.3.3). A build without the order check (NO_ECC_CHECK_PUBKEY_ORDER) still checks range and curve.
int	**wc_ecc_sign_hash**(const byte * in, word32 inlen, byte * out, word32 * outlen, WC_RNG * rng, ecc_key * key)This function signs a message digest using an ecc_key object to guarantee authenticity.
int	**wc_ecc_sign_hash_ex**(const byte * in, word32 inlen, WC_RNG * rng, ecc_key * key, mp_int * r, mp_int * s)Sign a message digest.
int	**wc_ecc_verify_hash**(const byte * sig, word32 siglen, const byte * hash, word32 hashlen, int * res, ecc_key * key)This function verifies the ECC signature of a hash to ensure authenticity. It returns the answer through res, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.
int	**wc_ecc_verify_hash_ex**(mp_int * r, mp_int * s, const byte * hash, word32 hashlen, int * res, ecc_key * key)Verify an ECC signature. Result is written to res. 1 is valid, 0 is invalid. Note: Do not use the return value to test for valid. Only use res.
int	**wc_ecc_init**(ecc_key * key)This function initializes an ecc_key object for future use with message verification or key negotiation.
int	**wc_ecc_init_ex**(ecc_key * key, void * heap, int devId)This function initializes an ecc_key object for future use with message verification or key negotiation.
ecc_key *	**wc_ecc_key_new**(void * heap)This function uses a user defined heap and allocates space for the key structure.
ecc_key *	**wc_ecc_key_new_ex**(void * heap, int devId)This function uses a user defined heap and allocates space for the key structure.
int	**wc_ecc_free**(ecc_key * key)This function frees an ecc_key object after it has been used.
void	**wc_ecc_fp_free**(void)This function frees the fixed_point cache, which can be used with ecc to speed up computation times. To use this functionality, FP_ECC (fixed_point ecc), should be defined. Threaded applications should call this function before exiting the thread.

	Name
int	**wc_ecc_is_valid_idx**(int n)Checks if an ECC idx is valid.
ecc_point *	**wc_ecc_new_point**(void)Allocate a new ECC point.
void	**wc_ecc_del_point**(ecc_point * p)Free an ECC point from memory.
int	**wc_ecc_copy_point**(const ecc_point * p, ecc_point * r)Copy the value of one point to another one.
int	**wc_ecc_cmp_point**(ecc_point * a, ecc_point * b)Compare the value of a point with another one.
int	**wc_ecc_point_is_at_infinity**(ecc_point * p)Checks if a point is at infinity. Returns 1 if point is at infinity, 0 if not, < 0 on error.
int	**wc_ecc_mulmod**(const mp_int * k, ecc_point * G, ecc_point * R, mp_int * a, mp_int * modulus, int map)Perform ECC Fixed Point multiplication.
int	**wc_ecc_export_x963**(ecc_key * key, byte * out, word32 * outLen)This function exports the ECC key from the ecc_key structure, storing the result in out. The key will be stored in ANSI X9.63 format. It stores the bytes written to the output buffer in outLen.
int	**wc_ecc_export_x963_ex**(ecc_key * key, byte * out, word32 * outLen, int compressed)This function exports the public key from the ecc_key structure, storing the result in out. The key will be stored in ANSI X9.63 format. It stores the bytes written to the output buffer in outLen. This function allows the additional option of compressing the certificate through the compressed parameter. When this parameter is true, the key will be stored in ANSI X9.63 compressed format.
int	**wc_ecc_import_x963**(const byte * in, word32 inLen, ecc_key * key)This function imports a public ECC key from a buffer containing the key stored in ANSI X9.63 format. This function will handle both compressed and uncompressed keys, as long as compressed keys are enabled at compile time through the HAVE_COMP_KEY option.

	Name
int	**wc_ecc_import_private_key**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ecc_key * key)This function imports a public/private ECC key pair from a buffer containing the raw private key, and a second buffer containing the ANSI X9.63 formatted public key. This function will handle both compressed and uncompressed keys, as long as compressed keys are enabled at compile time through the HAVE_COMP_KEY option.
int	**wc_ecc_import_wrapped_private**(ecc_key * key, int curve_id, const byte * seed, word32 seedSz, const byte * wrapped, word32 wrappedLen, word32 plainLen)This function imports an STM32 DHUK_protected private key onto an ecc_key for transparent hardware signing. The private scalar is supplied as a chip_bound wrapped blob together with the 256_bit derivation seed; the plaintext scalar is never imported. The key must be bound to the STM32 DHUK crypto_callback device (init with wc_ecc_init_ex(&key, heap, WC_DHUK_DEVID) after registering the device with wc_Stm32_DhukRegister). The curve is set from curve_id, so the key is ready to sign on return. Available only on STM32 builds with WOLFSSL_DHUK and a DHUK_capable SAES (WC_STM32_HAS_DHUK).
int	**wc_ecc_import_wrapped_private_ex**(ecc_key * key, int curve_id, const byte * wrapped, word32 wrappedLen, const byte * iv, word32 ivLen, const byte * tag, word32 tagLen, const byte * pub, word32 pubLen)This function restores a previously provisioned STM32 CCB_protected ECDSA key onto an ecc_key. The device_bound key is supplied as the wrapped scalar blob plus its AES_GCM iv/tag and the in_clear public key; signing is performed transparently with the scalar unwrapped SAES_>PKA in hardware. The key must be bound to the STM32 DHUK/CCB crypto_callback device (init with wc_ecc_init_ex(&key, heap, WC_DHUK_DEVID)). Available only on STM32 builds with WOLFSSL_DHUK and WOLFSSL_STM32_CCB.
int	**wc_ecc_rs_to_sig**(const char * r, const char * s, byte * out, word32 * outlen)This function converts the R and S portions of an ECC signature into a DER-encoded ECDSA signature. This function also stores the length written to the output buffer, out, in outlen.

	Name
int	**wc_ecc_import_raw**(ecc_key * key, const char * qx, const char * qy, const char * d, const char * curveName)This function fills an ecc_key structure with the raw components of an ECC key.
int	**wc_ecc_export_private_only**(ecc_key * key, byte * out, word32 * outLen)This function exports only the private key from an ecc_key structure. It stores the private key in the buffer out, and sets the bytes written to this buffer in outLen.
int	**wc_ecc_export_point_der**(const int curve_idx, ecc_point * point, byte * out, word32 * outLen)Export point to der.
int	**wc_ecc_import_point_der**(const byte * in, word32 inLen, const int curve_idx, ecc_point * point)Import point from der format.
int	**wc_ecc_size**(ecc_key * key)This function returns the key size of an ecc_key structure in octets.
int	**wc_ecc_sig_size_calc**(int sz)This function returns the worst case size for an ECC signature, given by: (keySz * 2) + SIG_HEADER_SZ + ECC_MAX_PAD_SZ. The actual signature size can be computed with wc_ecc_sign_hash.
int	**wc_ecc_sig_size**(const ecc_key * key)This function returns the worst case size for an ECC signature, given by: (keySz * 2) + SIG_HEADER_SZ + ECC_MAX_PAD_SZ. The actual signature size can be computed with wc_ecc_sign_hash.
ecEncCtx *	**wc_ecc_ctx_new**(int flags, WC_RNG * rng)This function allocates and initializes space for a new ECC context object to allow secure message exchange with ECC.
void	**wc_ecc_ctx_free**(ecEncCtx * ctx)This function frees the ecEncCtx object used for encrypting and decrypting messages.
int	**wc_ecc_ctx_reset**(ecEncCtx * ctx, WC_RNG * rng)This function resets an ecEncCtx structure to avoid having to free and allocate a new context object.
int	**wc_ecc_ctx_set_algo**(ecEncCtx * ctx, byte encAlgo, byte kdfAlgo, byte macAlgo)This function can optionally be called after wc_ecc_ctx_new. It sets the encryption, KDF, and MAC algorithms into an ecEncCtx object.

	Name
int	**wc_ecc_ctx_set_dev_id**(ecEncCtx * ctx, int devId)This function picks the device that ECIES operations using this context run on. Only available when WOLF_CRYPTO_CB is defined. A context starts at INVALID_DEVID, meaning software: ECIES does not copy the device from the private key, so this must be called for a crypto callback to be reached. The value is used both for the whole_operation ECIES callback and for the KDF, AES and HMAC steps of the software path. Passing a NULL context to wc_ecc_encrypt() or wc_ecc_decrypt() always means software. When WOLF_CRYPTO_CB_FIND is defined, an unset device id still goes through the registered finder, as it does for every other wolfCrypt operation. The setting is kept across wc_ecc_ctx_reset().
int	**wc_ecc_ctx_get_dev_id**(ecEncCtx * ctx, int * devId)This function reads back the device id set with wc_ecc_ctx_set_dev_id(). Crypto callback code can use it to learn which device it was called for. Only available when WOLF_CRYPTO_CB is defined. A context that was never given a device reads back INVALID_DEVID.
const byte *	**wc_ecc_ctx_get_own_salt**(ecEncCtx * ctx)This function returns the salt of an ecEncCtx object. This function should only be called when the ecEncCtx's state is ecSRV_INIT or ecCLI_INIT.
int	**wc_ecc_ctx_set_peer_salt**(ecEncCtx * ctx, const byte * salt)This function sets the peer salt of an ecEncCtx object.
int	**wc_ecc_ctx_set_kdf_salt**(ecEncCtx * ctx, const byte * salt, word32 sz)This function sets the salt pointer and length to use with KDF into the ecEncCtx object.
int	**wc_ecc_ctx_set_info**(ecEncCtx * ctx, const byte * info, int sz)This function can optionally be called before or after wc_ecc_ctx_set_peer_salt. It sets optional information for an ecEncCtx object.

	Name
int	**wc_ecc_encrypt**(ecc_key * privKey, ecc_key * pubKey, const byte * msg, word32 msgSz, byte * out, word32 * outSz, ecEncCtx * ctx)This function encrypts the given input message from msg to out. This function takes an optional ctx object as parameter. When supplied, encryption proceeds based on the ecEncCtx's encAlgo, kdfAlgo, and macAlgo. If ctx is not supplied, processing completes with the default algorithms, ecAES_128_CBC, ecHKDF_SHA256 and ecHMAC_SHA256. This function requires that the messages are padded according to the encryption type specified by ctx.
int	**wc_ecc_encrypt_ex**(ecc_key * privKey, ecc_key * pubKey, const byte * msg, word32 msgSz, byte * out, word32 * outSz, ecEncCtx * ctx, int compressed)This function encrypts the given input message from msg to out. This function takes an optional ctx object as parameter. When supplied, encryption proceeds based on the ecEncCtx's encAlgo, kdfAlgo, and macAlgo. If ctx is not supplied, processing completes with the default algorithms, ecAES_128_CBC, ecHKDF_SHA256 and ecHMAC_SHA256. This function requires that the messages are padded according to the encryption type specified by ctx.
int	**wc_ecc_decrypt**(ecc_key * privKey, ecc_key * pubKey, const byte * msg, word32 msgSz, byte * out, word32 * outSz, ecEncCtx * ctx)This function decrypts the ciphertext from msg to out. This function takes an optional ctx object as parameter. When supplied, encryption proceeds based on the ecEncCtx's encAlgo, kdfAlgo, and macAlgo. If ctx is not supplied, processing completes with the default algorithms, ecAES_128_CBC, ecHKDF_SHA256 and ecHMAC_SHA256. This function requires that the messages are padded according to the encryption type specified by ctx.
int	**wc_ecc_set_nonblock**(ecc_key * key, ecc_nb_ctx_t * ctx)Enable ECC support for non_blocking operations. Supported for Single Precision (SP) math with the following build options: WOLFSSL_SP_NONBLOCK WOLFSSL_SP_SMALL WOLFSSL_SP_NO_MALLOC WC_ECC_NONBLOCK.
int	**wc_ecc_set_curve**(ecc_key * key, int keysize, int curve_id)Compare a curve which has larger key than specified size or the curve matched curve ID, set a curve with smaller key size to the key.

	Name
mp_int *	**wc_ecc_key_get_priv**(ecc_key * key)Gets private key mp_int from ECC key.
size_t	**wc_ecc_get_sets_count**(void)Returns number of supported ECC curve sets.
const char *	**wc_ecc_get_name**(int curve_id)Gets curve name from curve ID.
int	**wc_ecc_make_key_ex2**(WC_RNG * rng, int keysize, ecc_key * key, int curve_id, int flags)Makes ECC key with extended options.
int	**wc_ecc_is_point**(ecc_point * ecp, mp_int * a, mp_int * b, mp_int * prime)Checks if point is on curve.
int	**wc_ecc_get_generator**(ecc_point * ecp, int curve_idx)Gets generator point for curve.
int	**wc_ecc_set_deterministic**(ecc_key * key, byte flag)Sets deterministic signing mode.
int	**wc_ecc_set_deterministic_ex**(ecc_key * key, byte flag, enum wc_HashType hashType)Sets deterministic signing with hash type.
int	**wc_ecc_gen_deterministic_k**(const byte * hash, word32 hashSz, enum wc_HashType hashType, mp_int * priv, mp_int * k, mp_int * order, void * heap)Generates deterministic k value for signing.
int	**wc_ecc_sign_set_k**(const byte * k, word32 klen, ecc_key * key)Sets k value for signing.
int	**wc_ecc_init_id**(ecc_key * key, unsigned char * id, int len, void * heap, int devId)Initializes ECC key with ID.
int	**wc_ecc_init_label**(ecc_key * key, const char * label, void * heap, int devId)Initializes ECC key with label.
int	**wc_ecc_set_flags**(ecc_key * key, word32 flags)Sets flags on ECC key.
void	**wc_ecc_fp_init**(void)Initializes fixed-point cache.
int	**wc_ecc_set_rng**(ecc_key * key, WC_RNG * rng)Sets RNG for ECC key.
int	**wc_ecc_get_curve_idx**(int curve_id)Gets curve index from curve ID.
int	**wc_ecc_get_curve_id**(int curve_idx)Gets curve ID from curve index.
int	**wc_ecc_get_curve_size_from_id**(int curve_id)Gets curve size from curve ID.
int	**wc_ecc_get_curve_idx_from_name**(const char * curveName)Gets curve index from curve name.
int	**wc_ecc_get_curve_size_from_name**(const char * curveName)Gets curve size from curve name.
int	**wc_ecc_get_curve_id_from_name**(const char * curveName)Gets curve ID from curve name.

	Name
int	**wc_ecc_get_curve_id_from_params**(int fieldSize, const byte * prime, word32 primeSz, const byte * Af, word32 AfSz, const byte * Bf, word32 BfSz, const byte * order, word32 orderSz, const byte * Gx, word32 GxSz, const byte * Gy, word32 GySz, int cofactor)Gets curve ID from curve parameters.
int	**wc_ecc_get_curve_id_from_dp_params**(const ecc_set_type * dp)Gets curve ID from domain parameters.
int	**wc_ecc_get_curve_id_from_oid**(const byte * oid, word32 len)Gets curve ID from OID.
const ecc_set_type *	**wc_ecc_get_curve_params**(int curve_idx)Gets curve parameters from curve index.
ecc_point *	**wc_ecc_new_point_h**(void * h)Allocates new ECC point with heap hint.
void	**wc_ecc_del_point_h**(ecc_point * p, void * h)Frees ECC point with heap hint.
void	**wc_ecc_forcezero_point**(ecc_point * p)Securely zeros ECC point.
int	**wc_ecc_point_is_on_curve**(ecc_point * p, int curve_idx)Checks if point is on curve.
int	**wc_ecc_import_x963_ex**(const byte * in, word32 inLen, ecc_key * key, int curve_id)Imports X9.63 format with curve ID.
int	**wc_ecc_import_private_key_ex**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ecc_key * key, int curve_id)Imports private key with curve ID.
int	**wc_ecc_rs_raw_to_sig**(const byte * r, word32 rSz, const byte * s, word32 sSz, byte * out, word32 * outlen)Converts raw r,s to signature.
int	**wc_ecc_sig_to_rs**(const byte * sig, word32 sigLen, byte * r, word32 * rLen, byte * s, word32 * sLen)Converts signature to raw r,s.
int	**wc_ecc_import_raw_ex**(ecc_key * key, const char * qx, const char * qy, const char * d, int curve_id)Imports raw key with curve ID.
int	**wc_ecc_import_unsigned**(ecc_key * key, const byte * qx, const byte * qy, const byte * d, int curve_id)Imports unsigned key with curve ID.
int	**wc_ecc_export_ex**(ecc_key * key, byte * qx, word32 * qxLen, byte * qy, word32 * qyLen, byte * d, word32 * dLen, int encType)Exports key with encoding type.
int	**wc_ecc_export_public_raw**(ecc_key * key, byte * qx, word32 * qxLen, byte * qy, word32 * qyLen)Exports public key in raw format.
int	**wc_ecc_export_private_raw**(ecc_key * key, byte * qx, word32 * qxLen, byte * qy, word32 * qyLen, byte * d, word32 * dLen)Exports private key in raw format.

	Name
int	**wc_ecc_export_point_der_ex**(const int curve_idx, ecc_point * point, byte * out, word32 * outLen, int compressed)Exports point in DER format with compression.
int	**wc_ecc_import_point_der_ex**(const byte * in, word32 inLen, const int curve_idx, ecc_point * point, int shortKeySize)Imports point from DER format.
int	**wc_ecc_get_oid**(word32 oidSum, const byte ** oid, word32 * oidSz)Gets OID for curve.
int	**wc_ecc_set_custom_curve**(ecc_key * key, const ecc_set_type * dp)Sets custom curve parameters.
ecEncCtx *	**wc_ecc_ctx_new_ex**(int flags, WC_RNG * rng, void * heap)Creates new ECC encryption context with heap.
int	**wc_ecc_ctx_set_own_salt**(ecEncCtx * ctx, const byte * salt, word32 sz)Sets own salt in context.
int	**wc_X963_KDF**(enum wc_HashType type, const byte * secret, word32 secretSz, const byte * sinfo, word32 sinfoSz, byte * out, word32 outSz)X9.63 Key Derivation Function.
int	**wc_ecc_curve_cache_init**(void)Initializes curve cache.
void	**wc_ecc_curve_cache_free**(void)Frees curve cache.
int	**wc_ecc_gen_k**(WC_RNG * rng, int size, mp_int * k, mp_int * order)Generates random k value.
int	**wc_ecc_set_handle**(ecc_key * key, remote_handle64 handle)Sets remote handle for hardware.
int	**wc_ecc_use_key_id**(ecc_key * key, word32 keyId, word32 flags)Uses key ID for hardware.
int	**wc_ecc_get_key_id**(ecc_key * key, word32 * keyId)Gets key ID from hardware key.

B.24.2 Functions Documentation

```
int wc_EccPrivateKeyToDer(
    ecc_key * key,
    byte * output,
    word32 outLen
)
```

Encodes ECC private key to DER format.

Parameters:

- **key** ECC key structure with private key
- **output** Buffer for DER encoded private key
- **outLen** Size of output buffer

See: wc_EccPrivateKeyDecode

Return:

- Size on success
- negative on error

Example

```
ecc_key key;
byte der[1024];
int derSz = wc_EccPrivateKeyToDer(&key, der, sizeof(der));
```

```
int wc_EccKeyDerSize(
    ecc_key * key,
    int pub
)
```

Calculates DER encoded ECC key size.

Parameters:

- **key** ECC key structure
- **pub** Non-zero to include public key

See: wc_EccPrivateKeyToDer

Return:

- Size on success
- negative on error

Example

```
ecc_key key;
int derSz = wc_EccKeyDerSize(&key, 1);
```

```
int wc_EccPrivateKeyToPKCS8(
    ecc_key * key,
    byte * output,
    word32 * outLen
)
```

Encodes ECC private key to PKCS#8 format.

Parameters:

- **key** ECC key structure with private key

- **output** Buffer for PKCS#8 encoded key
- **outLen** Pointer to buffer size (in/out)

See: wc_EccPrivateKeyToDer

Return:

- Size on success
- negative on error

Example

```
ecc_key key;
byte pkcs8[1024];
word32 pkcs8Sz = sizeof(pkcs8);
int ret = wc_EccPrivateKeyToPKCS8(&key, pkcs8, &pkcs8Sz);
```

```
int wc_EccKeyToPKCS8(
    ecc_key * key,
    byte * output,
    word32 * outLen
)
```

Encodes ECC key pair to PKCS#8 format.

Parameters:

- **key** ECC key structure with key pair
- **output** Buffer for PKCS#8 encoded key
- **outLen** Pointer to buffer size (in/out)

See: wc_EccPrivateKeyToPKCS8

Return:

- Size on success
- negative on error

Example

```
ecc_key key;
byte pkcs8[1024];
word32 pkcs8Sz = sizeof(pkcs8);
int ret = wc_EccKeyToPKCS8(&key, pkcs8, &pkcs8Sz);
```

```
int wc_EccPublicKeyDerSize(
    ecc_key * key,
    int with_AlgCurve
)
```

Calculates DER encoded ECC public key size.

Parameters:

- **key** ECC key structure
- **with_AlgCurve** Include algorithm and curve if non-zero

See: wc_EccPublicKeyToDer

Return:

- Size on success
- negative on error

Example

```
ecc_key key;
int derSz = wc_EccPublicKeyDerSize(&key, 1);
```

```
int wc_ecc_make_key(
    WC_RNG * rng,
    int keysize,
    ecc_key * key
)
```

This function generates a new ecc_key and stores it in key.

Parameters:

- **rng** pointer to an initialized RNG object with which to generate the key
- **keysize** desired length for the ecc_key
- **key** pointer to the ecc_key for which to generate a key

See:

- wc_ecc_init
- wc_ecc_shared_secret

Return:

- 0 Returned on success.
- ECC_BAD_ARG_E Returned if rng or key evaluate to NULL
- BAD_FUNC_ARG Returned if the specified key size is not in the correct range of supported keys
- MEMORY_E Returned if there is an error allocating memory while computing the ecc key
- MP_INIT_E may be returned if there is an error while computing the ecc key
- MP_READ_E may be returned if there is an error while computing the ecc key
- MP_CMP_E may be returned if there is an error while computing the ecc key
- MP_INVMOD_E may be returned if there is an error while computing the ecc key
- MP_EXPTMOD_E may be returned if there is an error while computing the ecc key
- MP_MOD_E may be returned if there is an error while computing the ecc key

- MP_MUL_E may be returned if there is an error while computing the ecc key
- MP_ADD_E may be returned if there is an error while computing the ecc key
- MP_MULMOD_E may be returned if there is an error while computing the ecc key
- MP_TO_E may be returned if there is an error while computing the ecc key
- MP_MEM may be returned if there is an error while computing the ecc key

Example

```
ecc_key key;
wc_ecc_init(&key);
WC_RNG rng;
wc_InitRng(&rng);
wc_ecc_make_key(&rng, 32, &key); // initialize 32 byte ecc key
```

```
int wc_ecc_make_key_ex(
    WC_RNG * rng,
    int keysize,
    ecc_key * key,
    int curve_id
)
```

This function generates a new ecc_key and stores it in key.

Parameters:

- **key** Pointer to store the created key.
- **keysize** size of key to be created in bytes, set based on curveId
- **rng** Rng to be used in key creation
- **curve_id** Curve to use for key

See:

- wc_ecc_make_key
- wc_ecc_get_curve_size_from_id

Return:

- 0 Returned on success.
- ECC_BAD_ARG_E Returned if rng or key evaluate to NULL
- BAD_FUNC_ARG Returned if the specified key size is not in the correct range of supported keys
- MEMORY_E Returned if there is an error allocating memory while computing the ecc key
- MP_INIT_E may be returned if there is an error while computing the ecc key
- MP_READ_E may be returned if there is an error while computing the ecc key
- MP_CMP_E may be returned if there is an error while computing the ecc key
- MP_INVMOD_E may be returned if there is an error while computing the ecc key
- MP_EXPTMOD_E may be returned if there is an error while computing the ecc key
- MP_MOD_E may be returned if there is an error while computing the ecc key
- MP_MUL_E may be returned if there is an error while computing the ecc key
- MP_ADD_E may be returned if there is an error while computing the ecc key
- MP_MULMOD_E may be returned if there is an error while computing the ecc key

- MP_TO_E may be returned if there is an error while computing the ecc key
- MP_MEM may be returned if there is an error while computing the ecc key

Example

```
ecc_key key;
int ret;
WC_RNG rng;
wc_ecc_init(&key);
wc_InitRng(&rng);
int curveId = ECC_SECP521R1;
int keySize = wc_ecc_get_curve_size_from_id(curveId);
ret = wc_ecc_make_key_ex(&rng, keySize, &key, curveId);
if (ret != MP_OKAY) {
    // error handling
}
```

```
int wc_ecc_make_pub(
    ecc_key * key,
    ecc_point * pubOut
)
```

wc_ecc_make_pub computes the public component from an ecc_key with an existing private component. If pubOut is supplied, the computed public key is stored there, else it is stored in the supplied ecc_key public component slot.

Parameters:

- **key** Pointer to an ecc_key containing a valid private component
- **pubOut** Optional pointer to an ecc_point struct in which to store the computed public key

See:

- wc_ecc_make_pub_ex
- wc_ecc_make_key

Return:

- 0 Returned on success.
- ECC_BAD_ARG_E Returned if key is NULL
- BAD_FUNC_ARG Returned if the supplied key is not a valid ecc_key.
- MEMORY_E Returned if there is an error allocating memory while computing the public key
- MP_INIT_E may be returned if there is an error while computing the public key
- MP_READ_E may be returned if there is an error while computing the public key
- MP_CMP_E may be returned if there is an error while computing the public key
- MP_INVMOD_E may be returned if there is an error while computing the public key
- MP_EXPTMOD_E may be returned if there is an error while computing the public key
- MP_MOD_E may be returned if there is an error while computing the public key
- MP_MUL_E may be returned if there is an error while computing the public key
- MP_ADD_E may be returned if there is an error while computing the public key

- MP_MULMOD_E may be returned if there is an error while computing the public key
- MP_TO_E may be returned if there is an error while computing the public key
- MP_MEM may be returned if there is an error while computing the public key
- ECC_OUT_OF_RANGE_E may be returned if there is an error while computing the public key
- ECC_PRIV_KEY_E may be returned if there is an error while computing the public key
- ECC_INF_E may be returned if there is an error while computing the public key

```
int wc_ecc_make_pub_ex(
    ecc_key * key,
    ecc_point * pubOut,
    WC_RNG * rng
)
```

wc_ecc_make_pub_ex computes the public component from an ecc_key with an existing private component. If pubOut is supplied, the computed public key is stored there, else it is stored in the supplied ecc_key public component slot. The supplied rng, if non-NULL, is used to blind the private key value used in the computation.

Parameters:

- **key** Pointer to an ecc_key containing a valid private component
- **pubOut** Optional pointer to an ecc_point struct in which to store the computed public key
- **rng** Rng to be used in the public key computation

See:

- wc_ecc_make_pub
- wc_ecc_make_key
- wc_InitRng

Return:

- 0 Returned on success.
- ECC_BAD_ARG_E Returned if key is NULL
- BAD_FUNC_ARG Returned if the supplied key is not a valid ecc_key.
- MEMORY_E Returned if there is an error allocating memory while computing the public key
- MP_INIT_E may be returned if there is an error while computing the public key
- MP_READ_E may be returned if there is an error while computing the public key
- MP_CMP_E may be returned if there is an error while computing the public key
- MP_INVMOD_E may be returned if there is an error while computing the public key
- MP_EXPTMOD_E may be returned if there is an error while computing the public key
- MP_MOD_E may be returned if there is an error while computing the public key
- MP_MUL_E may be returned if there is an error while computing the public key
- MP_ADD_E may be returned if there is an error while computing the public key
- MP_MULMOD_E may be returned if there is an error while computing the public key
- MP_TO_E may be returned if there is an error while computing the public key
- MP_MEM may be returned if there is an error while computing the public key
- ECC_OUT_OF_RANGE_E may be returned if there is an error while computing the public key
- ECC_PRIV_KEY_E may be returned if there is an error while computing the public key
- ECC_INF_E may be returned if there is an error while computing the public key

```
int wc_ecc_check_key(
    ecc_key * key
)
```

Perform sanity checks on ecc key validity.

Parameters:

- **key** Pointer to key to check.

See: wc_ecc_point_is_at_infinity

Return:

- MP_OKAY Success, key is OK.
- BAD_FUNC_ARG Returns if key is NULL.
- ECC_INF_E Returns if wc_ecc_point_is_at_infinity returns 1.

Example

```
ecc_key key;
WC_RNG rng;
int check_result;
wc_ecc_init(&key);
wc_InitRng(&rng);
wc_ecc_make_key(&rng, 32, &key);
check_result = wc_ecc_check_key(&key);

if (check_result == MP_OKAY)
{
    // key check succeeded
}
else
{
    // key check failed
}
```

```
void wc_ecc_key_free(
    ecc_key * key
)
```

This function frees an ecc_key key after it has been used.

Parameters:

- **key** pointer to the ecc_key structure to free

See:

- wc_ecc_key_new

- wc_ecc_init_ex

Example

```
// initialize key and perform ECC operations
...
wc_ecc_key_free(&key);
```

```
int wc_ecc_shared_secret(
    ecc_key * private_key,
    ecc_key * public_key,
    byte * out,
    word32 * outlen
)
```

This function generates a new secret key using a local private key and a received public key. It stores this shared secret key in the buffer out and updates outlen to hold the number of bytes written to the output buffer.

Parameters:

- **private_key** pointer to the ecc_key structure containing the local private key
- **public_key** pointer to the ecc_key structure containing the received public key
- **out** pointer to an output buffer in which to store the generated shared secret key
- **outlen** pointer to the word32 object containing the length of the output buffer. Will be overwritten with the length written to the output buffer upon successfully generating a shared secret key

See:

- wc_ecc_init
- wc_ecc_make_key

Return:

- 0 Returned upon successfully generating a shared secret key
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL
- ECC_BAD_ARG_E Returned if the type of the private key given as argument, private_key, is not ECC_PRIVATEKEY, or if the public and private key types (given by ecc->dp) are not equivalent
- MEMORY_E Returned if there is an error generating a new ecc point
- BUFFER_E Returned if the generated shared secret key is too long to store in the provided buffer
- MP_INIT_E may be returned if there is an error while computing the shared key
- MP_READ_E may be returned if there is an error while computing the shared key
- MP_CMP_E may be returned if there is an error while computing the shared key
- MP_INVMOD_E may be returned if there is an error while computing the shared key
- MP_EXPTMOD_E may be returned if there is an error while computing the shared key
- MP_MOD_E may be returned if there is an error while computing the shared key
- MP_MUL_E may be returned if there is an error while computing the shared key
- MP_ADD_E may be returned if there is an error while computing the shared key
- MP_MULMOD_E may be returned if there is an error while computing the shared key

- MP_TO_E may be returned if there is an error while computing the shared key
- MP_MEM may be returned if there is an error while computing the shared key
- ECC_INF_E returned when the computed shared secret is the point at infinity

Example

```
ecc_key priv, pub;
WC_RNG rng;
byte secret[1024]; // can hold 1024 byte shared secret key
word32 secretSz = sizeof(secret);
int ret;

wc_InitRng(&rng); // initialize rng
wc_ecc_init(&priv); // initialize key
wc_ecc_make_key(&rng, 32, &priv); // make public/private key pair
// receive public key, and initialise into pub
ret = wc_ecc_shared_secret(&priv, &pub, secret, &secretSz);
// generate secret key
if ( ret != 0 ) {
    // error generating shared secret key
}
```

```
int wc_ecc_shared_secret_ex(
    ecc_key * private_key,
    ecc_point * point,
    byte * out,
    word32 * outlen
)
```

Create an ECC shared secret between private key and public point. The point is first given full public key validation on the private key's curve with wc_ecc_check_key() (SP 800_56Ar3 5.6.2.3.3). A build without the order check (NO_ECC_CHECK_PUBKEY_ORDER) still checks range and curve.

Parameters:

- **private_key** The private ECC key.
- **point** The point to use (public key).
- **out** Output destination of the shared secret. Conforms to EC-DH from ANSI X9.63.
- **outlen** Input the max size and output the resulting size of the shared secret.

See: wc_ecc_verify_hash_ex

Return:

- MP_OKAY Indicates success.
- BAD_FUNC_ARG Error returned when any arguments are null.
- ECC_BAD_ARG_E Error returned if private_key->type is not ECC_PRIVATEKEY or private_key->idx fails to validate.
- BUFFER_E Error when outlen is too small.
- MEMORY_E Error to create a new point.
- MP_VAL possible when an initialization failure occurs.

- MP_MEM possible when an initialization failure occurs.
- IS_POINT_E, ECC_OUT_OF_RANGE_E, ECC_INF_E or MP_VAL when the point fails validation (not on the curve, out of range, or the identity).

Example

```
ecc_key key;
ecc_point* point;
byte shared_secret[];
int secret_size;
int result;

point = wc_ecc_new_point();

result = wc_ecc_shared_secret_ex(&key, point,
&shared_secret, &secret_size);

if (result != MP_OKAY)
{
    // Handle error
}
```

```
int wc_ecc_sign_hash(
    const byte * in,
    word32 inlen,
    byte * out,
    word32 * outlen,
    WC_RNG * rng,
    ecc_key * key
)
```

This function signs a message digest using an ecc_key object to guarantee authenticity.

Parameters:

- **in** pointer to the buffer containing the message hash to sign
- **inlen** length of the message hash to sign
- **out** buffer in which to store the generated signature
- **outlen** max length of the output buffer. Will store the bytes written to out upon successfully generating a message signature
- **key** pointer to a private ECC key with which to generate the signature

See: wc_ecc_verify_hash

Return:

- 0 Returned upon successfully generating a signature for the message digest
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the output buffer is too small to store the generated signature
- ECC_BAD_ARG_E Returned if the input key is not a private key, or if the ECC OID is invalid
- RNG_FAILURE_E Returned if the rng cannot successfully generate a satisfactory key

- MP_INIT_E may be returned if there is an error while computing the message signature
- MP_READ_E may be returned if there is an error while computing the message signature
- MP_CMP_E may be returned if there is an error while computing the message signature
- MP_INVMOD_E may be returned if there is an error while computing the message signature
- MP_EXPTMOD_E may be returned if there is an error while computing the message signature
- MP_MOD_E may be returned if there is an error while computing the message signature
- MP_MUL_E may be returned if there is an error while computing the message signature
- MP_ADD_E may be returned if there is an error while computing the message signature
- MP_MULMOD_E may be returned if there is an error while computing the message signature
- MP_TO_E may be returned if there is an error while computing the message signature
- MP_MEM may be returned if there is an error while computing the message signature

Example

```
ecc_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[512]; // will hold generated signature
sigSz = sizeof(sig);
byte digest[] = { // initialize with message hash };
wc_InitRng(&rng); // initialize rng
wc_ecc_init(&key); // initialize key
wc_ecc_make_key(&rng, 32, &key); // make public/private key pair
ret = wc_ecc_sign_hash(digest, sizeof(digest), sig, &sigSz, &key);
if ( ret != 0 ) {
    // error generating message signature
}
```

```
int wc_ecc_sign_hash_ex(
    const byte * in,
    word32 inlen,
    WC_RNG * rng,
    ecc_key * key,
    mp_int * r,
    mp_int * s
)
```

Sign a message digest.

Parameters:

- **in** The message digest to sign.
- **inlen** The length of the digest.
- **rng** Pointer to WC_RNG struct.
- **key** A private ECC key.
- **r** The destination for r component of the signature.
- **s** The destination for s component of the signature.

See: wc_ecc_verify_hash_ex

Return:

- MP_OKAY Returned upon successfully generating a signature for the message digest
- ECC_BAD_ARG_E Returned if the input key is not a private key, or if the ECC IDX is invalid, or if any of the input parameters evaluate to NULL, or if the output buffer is too small to store the generated signature
- RNG_FAILURE_E Returned if the rng cannot successfully generate a satisfactory key
- MP_INIT_E may be returned if there is an error while computing the message signature
- MP_READ_E may be returned if there is an error while computing the message signature
- MP_CMP_E may be returned if there is an error while computing the message signature
- MP_INVMOD_E may be returned if there is an error while computing the message signature
- MP_EXPTMOD_E may be returned if there is an error while computing the message signature
- MP_MOD_E may be returned if there is an error while computing the message signature
- MP_MUL_E may be returned if there is an error while computing the message signature
- MP_ADD_E may be returned if there is an error while computing the message signature
- MP_MULMOD_E may be returned if there is an error while computing the message signature
- MP_TO_E may be returned if there is an error while computing the message signature
- MP_MEM may be returned if there is an error while computing the message signature

Example

```
ecc_key key;
WC_RNG rng;
int ret, sigSz;
mp_int r; // destination for r component of signature.
mp_int s; // destination for s component of signature.

byte sig[512]; // will hold generated signature
sigSz = sizeof(sig);
byte digest[] = { initialize with message hash };
wc_InitRng(&rng); // initialize rng
wc_ecc_init(&key); // initialize key
mp_init(&r); // initialize r component
mp_init(&s); // initialize s component
wc_ecc_make_key(&rng, 32, &key); // make public/private key pair
ret = wc_ecc_sign_hash_ex(digest, sizeof(digest), &rng, &key, &r, &s);

if ( ret != MP_OKAY ) {
    // error generating message signature
}
```

```
int wc_ecc_verify_hash(
    const byte * sig,
    word32 siglen,
    const byte * hash,
    word32 hashlen,
    int * res,
    ecc_key * key
)
```

This function verifies the ECC signature of a hash to ensure authenticity. It returns the answer through res, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

Parameters:

- **sig** pointer to the buffer containing the signature to verify
- **siglen** length of the signature to verify
- **hash** pointer to the buffer containing the hash of the message verified
- **hashlen** length of the hash of the message verified
- **res** pointer to the result of the verification. 1 indicates the message was successfully verified
- **key** pointer to a public ECC key with which to verify the signature

See:

- wc_ecc_sign_hash
- wc_ecc_verify_hash_ex

Return:

- 0 Returned upon successfully performing the signature verification. Note: This does not mean that the signature is verified. The authenticity information is stored instead in res
- BAD_FUNC_ARG Returned any of the input parameters evaluate to NULL
- MEMORY_E Returned if there is an error allocating memory
- MP_INIT_E may be returned if there is an error while computing the message signature
- MP_READ_E may be returned if there is an error while computing the message signature
- MP_CMP_E may be returned if there is an error while computing the message signature
- MP_INVMOD_E may be returned if there is an error while computing the message signature
- MP_EXPTMOD_E may be returned if there is an error while computing the message signature
- MP_MOD_E may be returned if there is an error while computing the message signature
- MP_MUL_E may be returned if there is an error while computing the message signature
- MP_ADD_E may be returned if there is an error while computing the message signature
- MP_MULMOD_E may be returned if there is an error while computing the message signature
- MP_TO_E may be returned if there is an error while computing the message signature
- MP_MEM may be returned if there is an error while computing the message signature

Example

```
ecc_key key;
int ret, verified = 0;

byte sig[1024] { initialize with received signature };
byte digest[] = { initialize with message hash };
// initialize key with received public key
ret = wc_ecc_verify_hash(sig, sizeof(sig), digest,sizeof(digest),
&verified, &key);
if ( ret != 0 ) {
    // error performing verification
} else if ( verified == 0 ) {
    // the signature is invalid
}
```

```
int wc_ecc_verify_hash_ex(
    mp_int * r,
```

```
    mp_int * s,
    const byte * hash,
    word32 hashlen,
    int * res,
    ecc_key * key
)
```

Verify an ECC signature. Result is written to res. 1 is valid, 0 is invalid. Note: Do not use the return value to test for valid. Only use res.

Parameters:

- **r** The signature R component to verify
- **s** The signature S component to verify
- **hash** The hash (message digest) that was signed
- **hashlen** The length of the hash (octets)
- **res** Result of signature, 1==valid, 0==invalid
- **key** The corresponding public ECC key

See: wc_ecc_verify_hash

Return:

- MP_OKAY If successful (even if the signature is not valid)
- ECC_BAD_ARG_E Returns if arguments are null or if key-idx is invalid.
- MEMORY_E Error allocating ints or points.

Example

```
mp_int r;
mp_int s;
int res;
byte hash[] = { Some hash }
ecc_key key;

if(wc_ecc_verify_hash_ex(&r, &s, hash, hashlen, &res, &key) == MP_OKAY)
{
    // Check res
}
```

```
int wc_ecc_init(
    ecc_key * key
)
```

This function initializes an ecc_key object for future use with message verification or key negotiation.

Parameters:

- **key** pointer to the ecc_key object to initialize

See:

- wc_ecc_make_key
- wc_ecc_free

Return:

- 0 Returned upon successfully initializing the ecc_key object
- MEMORY_E Returned if there is an error allocating memory

Example

```
ecc_key key;
wc_ecc_init(&key);
```

```
int wc_ecc_init_ex(
    ecc_key * key,
    void * heap,
    int devId
)
```

This function initializes an ecc_key object for future use with message verification or key negotiation.

Parameters:

- **key** pointer to the ecc_key object to initialize
- **heap** pointer to a heap identifier
- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID if not used

See:

- wc_ecc_make_key
- wc_ecc_free
- wc_ecc_init

Return:

- 0 Returned upon successfully initializing the ecc_key object
- MEMORY_E Returned if there is an error allocating memory

Example

```
ecc_key key;
wc_ecc_init_ex(&key, heap, devId);
```

```
ecc_key * wc_ecc_key_new(
    void * heap
)
```

This function uses a user defined heap and allocates space for the key structure.

Parameters:

- **heap** pointer to a heap identifier

See:

- wc_ecc_key_new_ex
- wc_ecc_make_key
- wc_ecc_key_free
- wc_ecc_init

Return:

- Non-null returned upon successfully allocating and initializing the ecc_key object
- NULL returned if there is an error allocating memory

Example

```
wc_ecc_key_new(&heap);
```

```
ecc_key * wc_ecc_key_new_ex(
    void * heap,
    int devId
)
```

This function uses a user defined heap and allocates space for the key structure.

Parameters:

- **heap** pointer to a heap identifier
- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID if not used

See:

- wc_ecc_key_new
- wc_ecc_make_key
- wc_ecc_key_free
- wc_ecc_init

Return:

- Non-null returned upon successfully allocating and initializing the ecc_key object
- NULL returned if there is an error allocating memory

Example

```
wc_ecc_key_new_ex(&heap, MY_DEVID);
```

```
int wc_ecc_free(
    ecc_key * key
)
```

This function frees an ecc_key object after it has been used.

Parameters:

- **key** pointer to the ecc_key object to free

See: wc_ecc_init

Return: int integer returned indicating wolfSSL error or success status.

Example

```
// initialize key and perform secure exchanges
...
wc_ecc_free(&key);
```

```
void wc_ecc_fp_free(
    void
)
```

This function frees the fixed-point cache, which can be used with ecc to speed up computation times. To use this functionality, FP_ECC (fixed-point ecc), should be defined. Threaded applications should call this function before exiting the thread.

Parameters:

- **none** No parameters.

See: wc_ecc_free

Return: none No returns.

Example

```
ecc_key key;
// initialize key and perform secure exchanges
...

wc_ecc_fp_free();
```

```
int wc_ecc_is_valid_idx(
    int n
)
```

Checks if an ECC idx is valid.

Parameters:

- **n** The idx number to check.

See: none

Return:

- 1 Return if valid.
- 0 Return if not valid.

Example

```
ecc_key key;
WC_RNG rng;
int is_valid;
wc_ecc_init(&key);
wc_InitRng(&rng);
wc_ecc_make_key(&rng, 32, &key);
is_valid = wc_ecc_is_valid_idx(key.idx);
if (is_valid == 1)
{
    // idx is valid
}
else if (is_valid == 0)
{
    // idx is not valid
}
```

```
ecc_point * wc_ecc_new_point(
    void
)
```

Allocate a new ECC point.

Parameters:

- **none** No parameters.

See:

- wc_ecc_del_point
- wc_ecc_cmp_point
- wc_ecc_copy_point

- wc_ecc_del_point

Return:

- p A newly allocated point.
- NULL Returns NULL on error.
- ecc_point pointer on success
- NULL on failure

Allocates new ECC point.

Example

```
ecc_point* point;
point = wc_ecc_new_point();
if (point == NULL)
{
    // Handle point creation error
}
// Do stuff with point
```

Example

```
ecc_point* point = wc_ecc_new_point();
if (point != NULL) {
    // use point
    wc_ecc_del_point(point);
}
```

```
void wc_ecc_del_point(
    ecc_point * p
)
```

Free an ECC point from memory.

Parameters:

- **p** The point to free.

See:

- wc_ecc_new_point
- wc_ecc_cmp_point
- wc_ecc_copy_point

Return: none No returns.

Example

```
ecc_point* point;
point = wc_ecc_new_point();
if (point == NULL)
{
    // Handle point creation error
}
// Do stuff with point
wc_ecc_del_point(point);
```

```
int wc_ecc_copy_point(
    const ecc_point * p,
    ecc_point * r
)
```

Copy the value of one point to another one.

Parameters:

- **p** The point to copy.
- **r** The created point.

See:

- wc_ecc_new_point
- wc_ecc_cmp_point
- wc_ecc_del_point

Return:

- ECC_BAD_ARG_E Error thrown when p or r is null.
- MP_OKAY Point copied successfully
- ret Error from internal functions. Can be...

Example

```
ecc_point* point;
ecc_point* copied_point;
int copy_return;

point = wc_ecc_new_point();
copy_return = wc_ecc_copy_point(point, copied_point);
if (copy_return != MP_OKAY)
{
    // Handle error
}
```

```
int wc_ecc_cmp_point(
    ecc_point * a,
    ecc_point * b
)
```

Compare the value of a point with another one.

Parameters:

- **a** First point to compare.
- **b** Second point to compare.

See:

- wc_ecc_new_point
- wc_ecc_del_point
- wc_ecc_copy_point

Return:

- BAD_FUNC_ARG One or both arguments are NULL.
- MP_EQ The points are equal.
- ret Either MP_LT or MP_GT and signifies that the points are not equal.

Example

```
ecc_point* point;
ecc_point* point_to_compare;
int cmp_result;

point = wc_ecc_new_point();
point_to_compare = wc_ecc_new_point();
cmp_result = wc_ecc_cmp_point(point, point_to_compare);
if (cmp_result == BAD_FUNC_ARG)
{
    // arguments are invalid
}
else if (cmp_result == MP_EQ)
{
    // Points are equal
}
else
{
    // Points are not equal
}
```

```
int wc_ecc_point_is_at_infinity(
    ecc_point * p
)
```

Checks if a point is at infinity. Returns 1 if point is at infinity, 0 if not, < 0 on error.

Parameters:

- **p** The point to check.

See:

- wc_ecc_new_point
- wc_ecc_del_point
- wc_ecc_cmp_point
- wc_ecc_copy_point

Return:

- 1 p is at infinity.
- 0 p is not at infinity.
- <0 Error.

Example

```
ecc_point* point;
int is_infinity;
point = wc_ecc_new_point();

is_infinity = wc_ecc_point_is_at_infinity(point);
if (is_infinity < 0)
{
    // Handle error
}
else if (is_infinity == 0)
{
    // Point is not at infinity
}
else if (is_infinity == 1)
{
    // Point is at infinity
}
```

```
int wc_ecc_mulmod(
    const mp_int * k,
    ecc_point * G,
    ecc_point * R,
    mp_int * a,
    mp_int * modulus,
    int map
)
```

Perform ECC Fixed Point multiplication.

Parameters:

- **k** The multiplicand.
- **G** Base point to multiply.
- **R** Destination of product.
- **a** ECC curve parameter a.
- **modulus** The modulus for the curve.
- **map** If non-zero maps the point back to affine coordinates, otherwise it's left in jacobian-montgomery form.

See: none

Return:

- MP_OKAY Returns on successful operation.
- MP_INIT_E Returned if there is an error initializing an integer for use with the multiple precision integer (mp_int) library.

Example

```
ecc_point* base;
ecc_point* destination;
// Initialize points
base = wc_ecc_new_point();
destination = wc_ecc_new_point();
// Setup other arguments
mp_int multiplicand;
mp_int modulus;
mp_int a;
int map;
int rc;
rc = wc_ecc_mulmod(&multiplicand, base, destination, &a, &modulus, map);
```

```
int wc_ecc_export_x963(
    ecc_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports the ECC key from the ecc_key structure, storing the result in out. The key will be stored in ANSI X9.63 format. It stores the bytes written to the output buffer in outLen.

Parameters:

- **key** pointer to the ecc_key object to export
- **out** pointer to the buffer in which to store the ANSI X9.63 formatted key
- **outLen** size of the output buffer. On successfully storing the key, will hold the bytes written to the output buffer

See:

- wc_ecc_export_x963_ex
- wc_ecc_import_x963
- wc_ecc_make_pub

Return:

- 0 Returned on successfully exporting the ecc_key
- LENGTH_ONLY_E Returned if the output buffer evaluates to NULL, but the other two input parameters are valid. Indicates that the function is only returning the length required to store the key
- ECC_BAD_ARG_E Returned if any of the input parameters are NULL, or the key is unsupported (has an invalid index)
- BUFFER_E Returned if the output buffer is too small to store the ecc key. If the output buffer is too small, the size needed will be returned in outLen
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- MP_INIT_E may be returned if there is an error processing the ecc_key
- MP_READ_E may be returned if there is an error processing the ecc_key
- MP_CMP_E may be returned if there is an error processing the ecc_key
- MP_INVMOD_E may be returned if there is an error processing the ecc_key
- MP_EXPTMOD_E may be returned if there is an error processing the ecc_key
- MP_MOD_E may be returned if there is an error processing the ecc_key
- MP_MUL_E may be returned if there is an error processing the ecc_key
- MP_ADD_E may be returned if there is an error processing the ecc_key
- MP_MULMOD_E may be returned if there is an error processing the ecc_key
- MP_TO_E may be returned if there is an error processing the ecc_key
- MP_MEM may be returned if there is an error processing the ecc_key

Example

```
int ret;
byte buff[1024];
word32 buffSz = sizeof(buff);

ecc_key key;
// initialize key, make key
ret = wc_ecc_export_x963(&key, buff, &buffSz);
if ( ret != 0) {
    // error exporting key
}
```

```
int wc_ecc_export_x963_ex(
    ecc_key * key,
    byte * out,
    word32 * outLen,
    int compressed
)
```

This function exports the public key from the ecc_key structure, storing the result in out. The key will be stored in ANSI X9.63 format. It stores the bytes written to the output buffer in outLen. This function allows the additional option of compressing the certificate through the compressed parameter. When this parameter is true, the key will be stored in ANSI X9.63 compressed format.

Parameters:

- **key** pointer to the ecc_key object to export
- **out** pointer to the buffer in which to store the ANSI X9.63 formatted public key
- **outLen** size of the output buffer. On successfully storing the public key, will hold the bytes written to the output buffer
- **compressed** indicator of whether to store the key in compressed format. 1==compressed, 0==uncompressed

See:

- wc_ecc_export_x963
- wc_ecc_import_x963
- wc_ecc_make_pub

Return:

- 0 Returned on successfully exporting the ecc_key public component
- ECC_PRIVATEKEY_ONLY Returned if the ecc_key public component is missing
- NOT_COMPILED_IN Returned if the HAVE_COMP_KEY was not enabled at compile time, but the key was requested in compressed format
- LENGTH_ONLY_E Returned if the output buffer evaluates to NULL, but the other two input parameters are valid. Indicates that the function is only returning the length required to store the public key
- ECC_BAD_ARG_E Returned if any of the input parameters are NULL, or the key is unsupported (has an invalid index)
- BUFFER_E Returned if the output buffer is too small to store the public key. If the output buffer is too small, the size needed will be returned in outLen
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- MP_INIT_E may be returned if there is an error processing the ecc_key
- MP_READ_E may be returned if there is an error processing the ecc_key
- MP_CMP_E may be returned if there is an error processing the ecc_key
- MP_INVMOD_E may be returned if there is an error processing the ecc_key
- MP_EXPTMOD_E may be returned if there is an error processing the ecc_key
- MP_MOD_E may be returned if there is an error processing the ecc_key
- MP_MUL_E may be returned if there is an error processing the ecc_key
- MP_ADD_E may be returned if there is an error processing the ecc_key
- MP_MULMOD_E may be returned if there is an error processing the ecc_key
- MP_TO_E may be returned if there is an error processing the ecc_key
- MP_MEM may be returned if there is an error processing the ecc_key

Example

```
int ret;
byte buff[1024];
word32 buffSz = sizeof(buff);
ecc_key key;
// initialize key, make key
ret = wc_ecc_export_x963_ex(&key, buff, &buffSz, 1);
if ( ret != 0) {
    // error exporting key
}
```

```
int wc_ecc_import_x963(
    const byte * in,
    word32 inLen,
    ecc_key * key
)
```

This function imports a public ECC key from a buffer containing the key stored in ANSI X9.63 format. This function will handle both compressed and uncompressed keys, as long as compressed keys are enabled at compile time through the HAVE_COMP_KEY option.

Parameters:

- **in** pointer to the buffer containing the ANSI x9.63 formatted ECC key
- **inLen** length of the input buffer
- **key** pointer to the ecc_key object in which to store the imported key

See:

- wc_ecc_export_x963
- wc_ecc_import_private_key

Return:

- 0 Returned on successfully importing the ecc_key
- NOT_COMPILED_IN Returned if the HAVE_COMP_KEY was not enabled at compile time, but the key is stored in compressed format
- ECC_BAD_ARG_E Returned if in or key evaluate to NULL, or the inLen is even (according to the x9.63 standard, the key must be odd)
- MEMORY_E Returned if there is an error allocating memory
- ASN_PARSE_E Returned if there is an error parsing the ECC key; may indicate that the ECC key is not stored in valid ANSI X9.63 format
- IS_POINT_E Returned if the public key exported is not a point on the ECC curve
- MP_INIT_E may be returned if there is an error processing the ecc_key
- MP_READ_E may be returned if there is an error processing the ecc_key
- MP_CMP_E may be returned if there is an error processing the ecc_key
- MP_INVMOD_E may be returned if there is an error processing the ecc_key
- MP_EXPTMOD_E may be returned if there is an error processing the ecc_key
- MP_MOD_E may be returned if there is an error processing the ecc_key
- MP_MUL_E may be returned if there is an error processing the ecc_key
- MP_ADD_E may be returned if there is an error processing the ecc_key
- MP_MULMOD_E may be returned if there is an error processing the ecc_key
- MP_TO_E may be returned if there is an error processing the ecc_key
- MP_MEM may be returned if there is an error processing the ecc_key

Example

```
int ret;
byte buff[] = { initialize with ANSI X9.63 formatted key };

ecc_key pubKey;
```

```
wc_ecc_init(&pubKey);

ret = wc_ecc_import_x963(buff, sizeof(buff), &pubKey);
if ( ret != 0) {
    // error importing key
}
```

```
int wc_ecc_import_private_key(
    const byte * priv,
    word32 privSz,
    const byte * pub,
    word32 pubSz,
    ecc_key * key
)
```

This function imports a public/private ECC key pair from a buffer containing the raw private key, and a second buffer containing the ANSI X9.63 formatted public key. This function will handle both compressed and uncompressed keys, as long as compressed keys are enabled at compile time through the HAVE_COMP_KEY option.

Parameters:

- **priv** pointer to the buffer containing the raw private key
- **privSz** size of the private key buffer
- **pub** pointer to the buffer containing the ANSI x9.63 formatted ECC public key
- **pubSz** length of the public key input buffer
- **key** pointer to the ecc_key object in which to store the imported private/public key pair

See:

- wc_ecc_export_x963
- wc_ecc_import_private_key

Return:

- 0 Returned on successfully importing the ecc_key NOT_COMPILED_IN Returned if the HAVE_COMP_KEY was not enabled at compile time, but the key is stored in compressed format
- ECC_BAD_ARG_E Returned if in or key evaluate to NULL, or the inLen is even (according to the x9.63 standard, the key must be odd)
- MEMORY_E Returned if there is an error allocating memory
- ASN_PARSE_E Returned if there is an error parsing the ECC key; may indicate that the ECC key is not stored in valid ANSI X9.63 format
- IS_POINT_E Returned if the public key exported is not a point on the ECC curve
- MP_INIT_E may be returned if there is an error processing the ecc_key
- MP_READ_E may be returned if there is an error processing the ecc_key
- MP_CMP_E may be returned if there is an error processing the ecc_key
- MP_INVMOD_E may be returned if there is an error processing the ecc_key
- MP_EXPTMOD_E may be returned if there is an error processing the ecc_key
- MP_MOD_E may be returned if there is an error processing the ecc_key
- MP_MUL_E may be returned if there is an error processing the ecc_key

- MP_ADD_E may be returned if there is an error processing the ecc_key
- MP_MULMOD_E may be returned if there is an error processing the ecc_key
- MP_TO_E may be returned if there is an error processing the ecc_key
- MP_MEM may be returned if there is an error processing the ecc_key

Example

```
int ret;
byte pub[] = { initialize with ANSI X9.63 formatted key };
byte priv[] = { initialize with the raw private key };

ecc_key key;
wc_ecc_init(&key);
ret = wc_ecc_import_private_key(priv, sizeof(priv), pub, sizeof(pub),
&key);
if ( ret != 0) {
    // error importing key
}
```

```
int wc_ecc_import_wrapped_private(
    ecc_key * key,
    int curve_id,
    const byte * seed,
    word32 seedSz,
    const byte * wrapped,
    word32 wrappedLen,
    word32 plainLen
)
```

This function imports an STM32 DHUK-protected private key onto an ecc_key for transparent hardware signing. The private scalar is supplied as a chip-bound wrapped blob together with the 256-bit derivation seed; the plaintext scalar is never imported. The key must be bound to the STM32 DHUK crypto-callback device (init with wc_ecc_init_ex(&key, heap, WC_DHUK_DEVID) after registering the device with wc_Stm32_DhukRegister). The curve is set from curve_id, so the key is ready to sign on return. Available only on STM32 builds with WOLFSSL_DHUK and a DHUK-capable SAES (WC_STM32_HAS_DHUK).

Parameters:

- **key** pointer to the ecc_key (bound to WC_DHUK_DEVID) to import into.
- **curve_id** curve the scalar belongs to, e.g. ECC_SECP256R1.
- **seed** pointer to the 256-bit (32-byte) per-key DHUK derivation seed.
- **seedSz** length of seed in bytes; must be 32.
- **wrapped** pointer to the DHUK-wrapped private scalar blob.
- **wrappedLen** length of the wrapped blob; a non-zero multiple of the AES block size, no larger than the on-key buffer.
- **plainLen** length in bytes of the plaintext scalar inside the blob; must equal the scalar size of curve_id (32 for P-256, 48 for P-384).

See:

- wc_ecc_import_wrapped_private_ex

- wc_ecc_sign_hash
- wc_ecc_init_ex

Return:

- 0 Returned on success.
- BAD_FUNC_ARG Returned if key, seed, or wrapped is NULL; if seedSz is not 32; if wrappedLen is zero or not a multiple of the AES block size; if wrappedLen exceeds the on-key blob buffer; if plainLen is zero or larger than wrappedLen; if wrappedLen is larger than plainLen padded to a full AES block; or if plainLen does not match the scalar size of curve_id.
- <0 An error from the curve lookup if curve_id is not supported.

Example

```
ecc_key key;
wc_Stm32_DhukRegister(WC_DHUK_DEVID);
wc_ecc_init_ex(&key, NULL, WC_DHUK_DEVID);
if (wc_ecc_import_wrapped_private(&key, ECC_SECP256R1, seed, 32, wrapped,
        wrappedLen, plainLen) == 0) {
    wc_ecc_sign_hash(hash, hashLen, sig, &sigLen, &rng, &key);
}
wc_ecc_free(&key);
```

```
int wc_ecc_import_wrapped_private_ex(
    ecc_key * key,
    int curve_id,
    const byte * wrapped,
    word32 wrappedLen,
    const byte * iv,
    word32 ivLen,
    const byte * tag,
    word32 tagLen,
    const byte * pub,
    word32 pubLen
)
```

This function restores a previously provisioned STM32 CCB-protected ECDSA key onto an ecc_key. The device-bound key is supplied as the wrapped scalar blob plus its AES-GCM iv/tag and the in-clear public key; signing is performed transparently with the scalar unwrapped SAES->PKA in hardware. The key must be bound to the STM32 DHUK/CCB crypto-callback device (init with wc_ecc_init_ex(&key, heap, WC_DHUK_DEVID)). Available only on STM32 builds with WOLFSSL_DHUK and WOLFSSL_STM32_CCB.

Parameters:

- **key** pointer to the ecc_key (bound to WC_DHUK_DEVID) to import into.
- **curve_id** the ECC curve id of the wrapped key (e.g. ECC_SECP256R1).
- **wrapped** pointer to the CCB wrapped private scalar blob.
- **wrappedLen** length of the wrapped blob, no larger than the on-key buffer.
- **iv** pointer to the 16-byte AES-GCM iv of the blob.
- **ivLen** length of iv in bytes; must be 16.
- **tag** pointer to the 16-byte AES-GCM authentication tag of the blob.

- **tagLen** length of tag in bytes; must be 16.
- **pub** pointer to the public key in uncompressed qx||qy form.
- **pubLen** length of pub in bytes; must be twice the curve modulus size.

See:

- wc_ecc_import_wrapped_private
- wc_ecc_make_key_ex
- wc_ecc_sign_hash

Return:

- 0 Returned on success.
- BAD_FUNC_ARG Returned if key, wrapped, iv, tag, or pub is NULL; if ivLen or tagLen is not 16; if curve_id is not a supported curve; if wrappedLen is zero or exceeds the on-key blob buffer; or if pubLen is not twice the curve modulus size.
- <0 A negative error code may be returned if importing the public key fails.

Example

```
ecc_key key;
wc_Stm32_DhukRegister(WC_DHUK_DEVID);
wc_ecc_init_ex(&key, NULL, WC_DHUK_DEVID);
if (wc_ecc_import_wrapped_private_ex(&key, ECC_SECP256R1, wrapped,
        wrappedLen, iv, 16, tag, 16, pub, pubLen) == 0) {
    wc_ecc_sign_hash(hash, hashLen, sig, &sigLen, &rng, &key);
}
wc_ecc_free(&key);
```

```
int wc_ecc_rs_to_sig(
    const char * r,
    const char * s,
    byte * out,
    word32 * outlen
)
```

This function converts the R and S portions of an ECC signature into a DER-encoded ECDSA signature. This function also stores the length written to the output buffer, out, in outlen.

Parameters:

- **r** pointer to the buffer containing the R portion of the signature as a string
- **s** pointer to the buffer containing the S portion of the signature as a string
- **out** pointer to the buffer in which to store the DER-encoded ECDSA signature
- **outlen** length of the output buffer available. Will store the bytes written to the buffer after successfully converting the signature to ECDSA format

See:

- wc_ecc_sign_hash

- wc_ecc_sig_size

Return:

- 0 Returned on successfully converting the signature
- ECC_BAD_ARG_E Returned if any of the input parameters evaluate to NULL, or if the input buffer is not large enough to hold the DER-encoded ECDSA signature
- MP_INIT_E may be returned if there is an error processing the ecc_key
- MP_READ_E may be returned if there is an error processing the ecc_key
- MP_CMP_E may be returned if there is an error processing the ecc_key
- MP_INVMOD_E may be returned if there is an error processing the ecc_key
- MP_EXPTMOD_E may be returned if there is an error processing the ecc_key
- MP_MOD_E may be returned if there is an error processing the ecc_key
- MP_MUL_E may be returned if there is an error processing the ecc_key
- MP_ADD_E may be returned if there is an error processing the ecc_key
- MP_MULMOD_E may be returned if there is an error processing the ecc_key
- MP_TO_E may be returned if there is an error processing the ecc_key
- MP_MEM may be returned if there is an error processing the ecc_key

Example

```
int ret;
ecc_key key;
// initialize key, generate R and S

char r[] = { initialize with R };
char s[] = { initialize with S };
byte sig[wc_ecc_sig_size(key)];
// signature size will be 2 * ECC key size + ~10 bytes for ASN.1 overhead
word32 sigSz = sizeof(sig);
ret = wc_ecc_rs_to_sig(r, s, sig, &sigSz);
if ( ret != 0) {
    // error converting parameters to signature
}
```

```
int wc_ecc_import_raw(
    ecc_key * key,
    const char * qx,
    const char * qy,
    const char * d,
    const char * curveName
)
```

This function fills an ecc_key structure with the raw components of an ECC key.

Parameters:

- **key** pointer to an ecc_key structure to fill
- **qx** pointer to a buffer containing the x component of the base point as an ASCII hex string
- **qy** pointer to a buffer containing the y component of the base point as an ASCII hex string
- **d** pointer to a buffer containing the private key as an ASCII hex string

- **curveName** pointer to a string containing the ECC curve name, as found in ecc_sets

See: wc_ecc_import_private_key

Return:

- 0 Returned upon successfully importing into the ecc_key structure
- ECC_BAD_ARG_E Returned if any of the input values evaluate to NULL
- MEMORY_E Returned if there is an error initializing space to store the parameters of the ecc_key
- ASN_PARSE_E Returned if the input curveName is not defined in ecc_sets
- MP_INIT_E may be returned if there is an error processing the input parameters
- MP_READ_E may be returned if there is an error processing the input parameters
- MP_CMP_E may be returned if there is an error processing the input parameters
- MP_INVMOD_E may be returned if there is an error processing the input parameters
- MP_EXPTMOD_E may be returned if there is an error processing the input parameters
- MP_MOD_E may be returned if there is an error processing the input parameters
- MP_MUL_E may be returned if there is an error processing the input parameters
- MP_ADD_E may be returned if there is an error processing the input parameters
- MP_MULMOD_E may be returned if there is an error processing the input parameters
- MP_TO_E may be returned if there is an error processing the input parameters
- MP_MEM may be returned if there is an error processing the input parameters

Note: This function does not check that the imported public point lies on the curve. Define WOLFSSL_VALIDATE_ECC_IMPORT to validate the point on import, or call wc_ecc_check_key before the key is used.

Example

```
int ret;
ecc_key key;
wc_ecc_init(&key);

char qx[] = { initialize with x component of base point };
char qy[] = { initialize with y component of base point };
char d[]  = { initialize with private key };
ret = wc_ecc_import_raw(&key,qx, qy, d, "ECC-256");
if ( ret != 0) {
    // error initializing key with given inputs
}
```

```
int wc_ecc_export_private_only(
    ecc_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports only the private key from an ecc_key structure. It stores the private key in the buffer out, and sets the bytes written to this buffer in outLen.

Parameters:

- **key** pointer to an ecc_key structure from which to export the private key

- **out** pointer to the buffer in which to store the private key
- **outLen** pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the private key

See: wc_ecc_import_private_key

Return:

- 0 Returned upon successfully exporting the private key
- ECC_BAD_ARG_E Returned if any of the input values evaluate to NULL
- MEMORY_E Returned if there is an error initializing space to store the parameters of the ecc_key
- ASN_PARSE_E Returned if the input curveName is not defined in ecc_sets
- MP_INIT_E may be returned if there is an error processing the input parameters
- MP_READ_E may be returned if there is an error processing the input parameters
- MP_CMP_E may be returned if there is an error processing the input parameters
- MP_INVMOD_E may be returned if there is an error processing the input parameters
- MP_EXPTMOD_E may be returned if there is an error processing the input parameters
- MP_MOD_E may be returned if there is an error processing the input parameters
- MP_MUL_E may be returned if there is an error processing the input parameters
- MP_ADD_E may be returned if there is an error processing the input parameters
- MP_MULMOD_E may be returned if there is an error processing the input parameters
- MP_TO_E may be returned if there is an error processing the input parameters
- MP_MEM may be returned if there is an error processing the input parameters

Example

```
int ret;
ecc_key key;
// initialize key, make key

char priv[ECC_KEY_SIZE];
word32 privSz = sizeof(priv);
ret = wc_ecc_export_private_only(&key, priv, &privSz);
if ( ret != 0) {
    // error exporting private key
}
```

```
int wc_ecc_export_point_der(
    const int curve_idx,
    ecc_point * point,
    byte * out,
    word32 * outLen
)
```

Export point to der.

Parameters:

- **curve_idx** Index of the curve used from ecc_sets.
- **point** Point to export to der.
- **out** Destination for the output.

- **outLen** Maxsize allowed for output, destination for final size of output

See: wc_ecc_import_point_der

Return:

- 0 Returned on success.
- ECC_BAD_ARG_E Returns if curve_idx is less than 0 or invalid. Also returns when
- LENGTH_ONLY_E outLen is set but nothing else.
- BUFFER_E Returns if outLen is less than 1 + 2 * the curve size.
- MEMORY_E Returns if there is a problem allocating memory.

Example

```
int curve_idx;
ecc_point* point;
byte out[];
word32 outLen;
wc_ecc_export_point_der(curve_idx, point, out, &outLen);
```

```
int wc_ecc_import_point_der(
    const byte * in,
    word32 inLen,
    const int curve_idx,
    ecc_point * point
)
```

Import point from der format.

Parameters:

- **in** der buffer to import point from.
- **inLen** Length of der buffer.
- **curve_idx** Index of curve.
- **point** Destination for point.

See: wc_ecc_export_point_der

Return:

- ECC_BAD_ARG_E Returns if any arguments are null or if inLen is even.
- MEMORY_E Returns if there is an error initializing
- NOT_COMPILED_IN Returned if HAVE_COMP_KEY is not true and in is a compressed cert
- MP_OKAY Successful operation.

Example

```
byte in[];
word32 inLen;
int curve_idx;
```

```
ecc_point* point;
wc_ecc_import_point_der(in, inLen, curve_idx, point);
```

```
int wc_ecc_size(
    ecc_key * key
)
```

This function returns the key size of an ecc_key structure in octets.

Parameters:

- **key** pointer to an ecc_key structure for which to get the key size

See: wc_ecc_make_key

Return:

- Given a valid key, returns the key size in octets
- 0 Returned if the given key is NULL

Example

```
int keySz;
ecc_key key;
// initialize key, make key
keySz = wc_ecc_size(&key);
if ( keySz == 0) {
    // error determining key size
}
```

```
int wc_ecc_sig_size_calc(
    int sz
)
```

This function returns the worst case size for an ECC signature, given by: (keySz * 2) + SIG_HEADER_SZ + ECC_MAX_PAD_SZ. The actual signature size can be computed with wc_ecc_sign_hash.

Parameters:

- **key** size

See:

- wc_ecc_sign_hash
- wc_ecc_sig_size

Return: returns the maximum signature size, in octets

Example

```
int sigSz = wc_ecc_sig_size_calc(32);
if ( sigSz == 0) {
    // error determining sig size
}
```

```
int wc_ecc_sig_size(
    const ecc_key * key
)
```

This function returns the worst case size for an ECC signature, given by: (keySz * 2) + SIG_HEADER_SZ + ECC_MAX_PAD_SZ. The actual signature size can be computed with wc_ecc_sign_hash.

Parameters:

- **key** pointer to an ecc_key structure for which to get the signature size

See:

- wc_ecc_sign_hash
- wc_ecc_sig_size_calc

Return:

- Success Given a valid key, returns the maximum signature size, in octets
- 0 Returned if the given key is NULL

Example

```
int sigSz;
ecc_key key;
// initialize key, make key

sigSz = wc_ecc_sig_size(&key);
if ( sigSz == 0) {
    // error determining sig size
}
```

```
ecEncCtx * wc_ecc_ctx_new(
    int flags,
    WC_RNG * rng
)
```

This function allocates and initializes space for a new ECC context object to allow secure message exchange with ECC.

Parameters:

- **flags** indicate whether this is a server or client context Options are: REQ_RESP_CLIENT, and REQ_RESP_SERVER
- **rng** pointer to a RNG object with which to generate a salt
- **flags** Context flags
- **rng** Random number generator

See:

- wc_ecc_encrypt
- wc_ecc_encrypt_ex
- wc_ecc_decrypt
- wc_ecc_ctx_free

Return:

- Success On successfully generating a new ecEncCtx object, returns a pointer to that object
- NULL Returned if the function fails to generate a new ecEncCtx object
- ecEncCtx pointer on success
- NULL on failure

Creates new ECC encryption context.

Example

```
ecEncCtx* ctx;
WC_RNG rng;
wc_InitRng(&rng);
ctx = wc_ecc_ctx_new(REQ_RESP_CLIENT, &rng);
if(ctx == NULL) {
    // error generating new ecEncCtx object
}
```

Example

```
WC_RNG rng;
ecEncCtx* ctx = wc_ecc_ctx_new(0, &rng);
```

```
void wc_ecc_ctx_free(
    ecEncCtx * ctx
)
```

This function frees the ecEncCtx object used for encrypting and decrypting messages.

Parameters:

- **ctx** pointer to the ecEncCtx object to free

See: wc_ecc_ctx_new

Return: none Returns.

Example

```
ecEncCtx* ctx;
WC_RNG rng;
wc_InitRng(&rng);
ctx = wc_ecc_ctx_new(REQ_RESP_CLIENT, &rng);
// do secure communication
...
wc_ecc_ctx_free(&ctx);
```

```
int wc_ecc_ctx_reset(
    ecEncCtx * ctx,
    WC_RNG * rng
)
```

This function resets an ecEncCtx structure to avoid having to free and allocate a new context object.

Parameters:

- **ctx** pointer to the ecEncCtx object to reset
- **rng** pointer to an RNG object with which to generate a new salt
- **ctx** ECC encryption context
- **rng** Random number generator

See:

- wc_ecc_ctx_new
- wc_ecc_ctx_set_dev_id
- wc_ecc_ctx_new

Return:

- 0 Returned if the ecEncCtx structure is successfully reset
- BAD_FUNC_ARG Returned if either rng or ctx is NULL
- RNG_FAILURE_E Returned if there is an error generating a new salt for the ECC object
- 0 on success
- negative on error

Note: The device id set with wc_ecc_ctx_set_dev_id() (WOLF_CRYPTO_CB builds) is kept across the reset, like the heap hint.

Resets ECC encryption context.

Example

```
ecEncCtx* ctx;
WC_RNG rng;
wc_InitRng(&rng);
ctx = wc_ecc_ctx_new(REQ_RESP_CLIENT, &rng);
// do secure communication
...
```

```
wc_ecc_ctx_reset(&ctx, &rng);
// do more secure communication
```

Example

```
ecEncCtx* ctx;
WC_RNG rng;
int ret = wc_ecc_ctx_reset(ctx, &rng);
```

```
int wc_ecc_ctx_set_algo(
    ecEncCtx * ctx,
    byte encAlgo,
    byte kdfAlgo,
    byte macAlgo
)
```

This function can optionally be called after wc_ecc_ctx_new. It sets the encryption, KDF, and MAC algorithms into an ecEncCtx object.

Parameters:

- **ctx** pointer to the ecEncCtx for which to set the info
- **encAlgo** encryption algorithm to use.
- **kdfAlgo** KDF algorithm to use.
- **macAlgo** MAC algorithm to use.

See: wc_ecc_ctx_new

Return:

- 0 Returned upon successfully setting the information for the ecEncCtx object.
- BAD_FUNC_ARG Returned if the given ecEncCtx object is NULL.

Example

```
ecEncCtx* ctx;
// initialize ctx
if(wc_ecc_ctx_set_algo(&ctx, ecAES_128_CTR, ecHKDF_SHA256, ecHMAC_SHA256))) {
    // error setting info
}
```

```
int wc_ecc_ctx_set_dev_id(
    ecEncCtx * ctx,
    int devId
)
```

This function picks the device that ECIES operations using this context run on. Only available when WOLF_CRYPTO_CB is defined. A context starts at INVALID_DEVID, meaning software: ECIES does not copy the device from the private key, so this must be called for a crypto callback to be reached. The

value is used both for the whole-operation ECIES callback and for the KDF, AES and HMAC steps of the software path. Passing a NULL context to wc_ecc_encrypt() or wc_ecc_decrypt() always means software. When WOLF_CRYPTO_CB_FIND is defined, an unset device id still goes through the registered finder, as it does for every other wolfCrypt operation. The setting is kept across wc_ecc_ctx_reset().

Parameters:

- **ctx** pointer to the ecEncCtx for which to set the device id
- **devId** device id to use, or INVALID_DEVID for software

See:

- wc_ecc_ctx_get_dev_id
- wc_ecc_ctx_new
- wc_ecc_ctx_reset

Return:

- 0 Returned upon successfully setting the device id.
- BAD_FUNC_ARG Returned if the given context is NULL.

Example

```
ecEncCtx* ctx = wc_ecc_ctx_new(REQ_RESP_CLIENT, &rng);
if (wc_ecc_ctx_set_dev_id(ctx, myDevId) != 0) {
    // error setting device id
}
```

```
int wc_ecc_ctx_get_dev_id(
    ecEncCtx * ctx,
    int * devId
)
```

This function reads back the device id set with wc_ecc_ctx_set_dev_id(). Crypto callback code can use it to learn which device it was called for. Only available when WOLF_CRYPTO_CB is defined. A context that was never given a device reads back INVALID_DEVID.

Parameters:

- **ctx** pointer to the ecEncCtx to read the device id from
- **devId** pointer that receives the device id

See:

- wc_ecc_ctx_set_dev_id
- wc_ecc_ctx_new

Return:

- 0 Returned upon successfully reading the device id.
- BAD_FUNC_ARG Returned if the given context or output pointer is NULL.

Example

```
int devId;
if (wc_ecc_ctx_get_dev_id(ctx, &devId) != 0) {
    // error reading device id
}
```

```
const byte * wc_ecc_ctx_get_own_salt(
    ecEncCtx * ctx
)
```

This function returns the salt of an ecEncCtx object. This function should only be called when the ecEncCtx's state is ecSRV_INIT or ecCLI_INIT.

Parameters:

- **ctx** pointer to the ecEncCtx object from which to get the salt
- **ctx** ECC encryption context

See:

- wc_ecc_ctx_new
- wc_ecc_ctx_set_peer_salt
- wc_ecc_ctx_set_kdf_salt
- wc_ecc_ctx_set_own_salt

Return:

- Success On success, returns the ecEncCtx salt
- NULL Returned if the ecEncCtx object is NULL, or the ecEncCtx's state is not ecSRV_INIT or ecCLI_INIT. In the latter two cases, this function also sets the ecEncCtx's state to ecSRV_BAD_STATE or ecCLI_BAD_STATE, respectively
- Salt pointer on success
- NULL on failure

Gets own salt from context.

Example

```
ecEncCtx* ctx;
WC_RNG rng;
const byte* salt;
wc_InitRng(&rng);
ctx = wc_ecc_ctx_new(REQ_RESP_CLIENT, &rng);
salt = wc_ecc_ctx_get_own_salt(&ctx);
if(salt == NULL) {
```

```
    // error getting salt
}
```

Example

```
ecEncCtx* ctx;
const byte* salt = wc_ecc_ctx_get_own_salt(ctx);
```

```
int wc_ecc_ctx_set_peer_salt(
    ecEncCtx * ctx,
    const byte * salt
)
```

This function sets the peer salt of an ecEncCtx object.

Parameters:

- **ctx** pointer to the ecEncCtx for which to set the salt
- **salt** pointer to the peer's salt

See:

- wc_ecc_ctx_get_own_salt
- wc_ecc_ctx_set_kdf_salt

Return:

- 0 Returned upon successfully setting the peer salt for the ecEncCtx object.
- BAD_FUNC_ARG Returned if the given ecEncCtx object is NULL or has an invalid protocol, or if the given salt is NULL
- BAD_ENC_STATE_E Returned if the ecEncCtx's state is ecSRV_SALT_GET or ecCLI_SALT_GET. In the latter two cases, this function also sets the ecEncCtx's state to ecSRV_BAD_STATE or ecCLI_BAD_STATE, respectively

Example

```
ecEncCtx* cliCtx, srvCtx;
WC_RNG rng;
const byte* cliSalt, srvSalt;
int ret;

wc_InitRng(&rng);
cliCtx = wc_ecc_ctx_new(REQ_RESP_CLIENT, &rng);
srvCtx = wc_ecc_ctx_new(REQ_RESP_SERVER, &rng);

cliSalt = wc_ecc_ctx_get_own_salt(&cliCtx);
srvSalt = wc_ecc_ctx_get_own_salt(&srvCtx);
ret = wc_ecc_ctx_set_peer_salt(&cliCtx, srvSalt);
```

```
int wc_ecc_ctx_set_kdf_salt(
    ecEncCtx * ctx,
    const byte * salt,
    word32 sz
)
```

This function sets the salt pointer and length to use with KDF into the ecEncCtx object.

Parameters:

- **ctx** pointer to the ecEncCtx for which to set the salt
- **salt** pointer to salt buffer
- **sz** length salt in bytes

See:

- wc_ecc_ctx_get_own_salt
- wc_ecc_ctx_get_peer_salt

Return:

- 0 Returned upon successfully setting the salt for the ecEncCtx object.
- BAD_FUNC_ARG Returned if the given ecEncCtx object is NULL or if the given salt is NULL and length is not NULL.

Example

```
ecEncCtx* srvCtx;
WC_RNG rng;
byte cliSalt[] = { fixed salt data };
word32 cliSaltLen = (word32)sizeof(cliSalt);
int ret;

wc_InitRng(&rng);
cliCtx = wc_ecc_ctx_new(REQ_RESP_SERVER, &rng);

ret = wc_ecc_ctx_set_kdf_salt(&cliCtx, cliSalt, cliSaltLen);
```

```
int wc_ecc_ctx_set_info(
    ecEncCtx * ctx,
    const byte * info,
    int sz
)
```

This function can optionally be called before or after wc_ecc_ctx_set_peer_salt. It sets optional information for an ecEncCtx object.

Parameters:

- **ctx** pointer to the ecEncCtx for which to set the info

- **info** pointer to a buffer containing the info to set
- **sz** size of the info buffer

See: wc_ecc_ctx_new

Return:

- 0 Returned upon successfully setting the information for the ecEncCtx object.
- BAD_FUNC_ARG Returned if the given ecEncCtx object is NULL, the input info is NULL or it's size is invalid

Example

```
ecEncCtx* ctx;
byte info[] = { initialize with information };
// initialize ctx, get salt,
if(wc_ecc_ctx_set_info(&ctx, info, sizeof(info))) {
    // error setting info
}
```

```
int wc_ecc_encrypt(
    ecc_key * privKey,
    ecc_key * pubKey,
    const byte * msg,
    word32 msgSz,
    byte * out,
    word32 * outSz,
    ecEncCtx * ctx
)
```

This function encrypts the given input message from msg to out. This function takes an optional ctx object as parameter. When supplied, encryption proceeds based on the ecEncCtx's encAlgo, kdfAlgo, and macAlgo. If ctx is not supplied, processing completes with the default algorithms, ecAES_128_CBC, ecHKDF_SHA256 and ecHMAC_SHA256. This function requires that the messages are padded according to the encryption type specified by ctx.

Parameters:

- **privKey** pointer to the ecc_key object containing the private key to use for encryption
- **pubKey** pointer to the ecc_key object containing the public key of the peer with whom one wishes to communicate
- **msg** pointer to the buffer holding the message to encrypt
- **msgSz** size of the buffer to encrypt
- **out** pointer to the buffer in which to store the encrypted ciphertext
- **outSz** pointer to a word32 object containing the available size in the out buffer. Upon successfully encrypting the message, holds the number of bytes written to the output buffer
- **ctx** Optional: pointer to an ecEncCtx object specifying different encryption algorithms to use

See:

- wc_ecc_encrypt_ex

- wc_ecc_decrypt
- wc_ecc_ctx_set_dev_id

Return:

- 0 Returned upon successfully encrypting the input message
- BAD_FUNC_ARG Returned if privKey, pubKey, msg, msgSz, out, or outSz are NULL, or the ctx object specifies an unsupported encryption type
- BAD_ENC_STATE_E Returned if the ctx object given is in a state that is not appropriate for encryption
- BUFFER_E Returned if the supplied output buffer is too small to store the encrypted ciphertext
- MEMORY_E Returned if there is an error allocating memory for the shared secret key

Note:

- Selecting an AES-GCM DEM algorithm (ecAES_128_GCM, ecAES_256_GCM) in the default IV mode requires the WOLFSSL_ECIES_STATIC_GCM_NONCE build macro; otherwise this function returns NOT_COMPILED_IN. See wc_ecc_encrypt_ex for the full rationale.
- The device this runs on comes from the context (wc_ecc_ctx_set_dev_id), not from privKey->devId. A NULL context, or one that was never given a device, runs in software.

Example

```
byte msg[] = { initialize with msg to encrypt. Ensure padded to block size };
byte out[sizeof(msg)];
word32 outSz = sizeof(out);
int ret;
ecc_key cli, serv;
// initialize cli with private key
// initialize serv with received public key

ecEncCtx* cliCtx, servCtx;
// initialize cliCtx and servCtx
// exchange salts
ret = wc_ecc_encrypt(&cli, &serv, msg, sizeof(msg), out, &outSz, cliCtx);
if(ret != 0) {
    // error encrypting message
}
```

```
int wc_ecc_encrypt_ex(
    ecc_key * privKey,
    ecc_key * pubKey,
    const byte * msg,
    word32 msgSz,
    byte * out,
    word32 * outSz,
    ecEncCtx * ctx,
    int compressed
)
```

This function encrypts the given input message from msg to out. This function takes an optional ctx object as parameter. When supplied, encryption proceeds based on the ecEncCtx's encAlgo, kdfAlgo, and macAlgo. If ctx is not supplied, processing completes with the default algorithms, ecAES_128_CBC, ecHKDF_SHA256 and ecHMAC_SHA256. This function requires that the messages are padded according to the encryption type specified by ctx.

Parameters:

- **privKey** pointer to the ecc_key object containing the private key to use for encryption
- **pubKey** pointer to the ecc_key object containing the public key of the peer with whom one wishes to communicate
- **msg** pointer to the buffer holding the message to encrypt
- **msgSz** size of the buffer to encrypt
- **out** pointer to the buffer in which to store the encrypted ciphertext
- **outSz** pointer to a word32 object containing the available size in the out buffer. Upon successfully encrypting the message, holds the number of bytes written to the output buffer
- **ctx** Optional: pointer to an ecEncCtx object specifying different encryption algorithms to use
- **compressed** Public key field is to be output in compressed format.

See:

- wc_ecc_encrypt
- wc_ecc_decrypt
- wc_ecc_ctx_set_dev_id

Return:

- 0 Returned upon successfully encrypting the input message
- BAD_FUNC_ARG Returned if privKey, pubKey, msg, msgSz, out, or outSz are NULL, or the ctx object specifies an unsupported encryption type
- BAD_ENC_STATE_E Returned if the ctx object given is in a state that is not appropriate for encryption
- BUFFER_E Returned if the supplied output buffer is too small to store the encrypted ciphertext
- MEMORY_E Returned if there is an error allocating memory for the shared secret key
- NOT_COMPILED_IN Returned if an AES-GCM DEM algorithm (ecAES_128_GCM or ecAES_256_GCM) is requested in the default IV mode without the WOLFSSL_ECIES_STATIC_GCM_NONCE build macro defined

Note:

- The AES-GCM DEM algorithms (ecAES_128_GCM, ecAES_256_GCM) in the default IV mode use a fixed all-zero GCM nonce. That is safe only because ECIES derives a fresh symmetric key from a fresh ephemeral key on every encryption, so the (key, nonce) pair never repeats; reusing the ephemeral key is catastrophic for AES-GCM. For that reason the fixed-nonce GCM DEM is off by default and must be enabled with the WOLFSSL_ECIES_STATIC_GCM_NONCE build macro, otherwise this function returns NOT_COMPILED_IN for a GCM algorithm. The macro is not needed with WOLFSSL_ECIES_GEN_IV (random per-message nonce) or WOLFSSL_ECIES_OLD (nonce derived from the KDF output).
- The device this runs on comes from the context (wc_ecc_ctx_set_dev_id), not from privKey->devId. A NULL context, or one that was never given a device, runs in software.

Example

```
byte msg[] = { initialize with msg to encrypt. Ensure padded to block size };
byte out[sizeof(msg)];
word32 outSz = sizeof(out);
int ret;
ecc_key cli, serv;
// initialize cli with private key
// initialize serv with received public key

ecEncCtx* cliCtx, servCtx;
// initialize cliCtx and servCtx
// exchange salts
ret = wc_ecc_encrypt_ex(&cli, &serv, msg, sizeof(msg), out, &outSz, cliCtx,
    1);
if(ret != 0) {
    // error encrypting message
}
```

```
int wc_ecc_decrypt(
    ecc_key * privKey,
    ecc_key * pubKey,
    const byte * msg,
    word32 msgSz,
    byte * out,
    word32 * outSz,
    ecEncCtx * ctx
)
```

This function decrypts the ciphertext from msg to out. This function takes an optional ctx object as parameter. When supplied, encryption proceeds based on the ecEncCtx's encAlgo, kdfAlgo, and macAlgo. If ctx is not supplied, processing completes with the default algorithms, ecAES_128_CBC, ecHKDF_SHA256 and ecHMAC_SHA256. This function requires that the messages are padded according to the encryption type specified by ctx.

Parameters:

- **privKey** pointer to the ecc_key object containing the private key to use for decryption
- **pubKey** only used when built with WOLFSSL_ECIES_OLD: pointer to the ecc_key object containing the public key of the peer with whom one wishes to communicate. In the default message format the sender's ephemeral public key is read from the start of msg instead and pubKey is ignored (it may be NULL and is left unmodified)
- **msg** pointer to the buffer holding the ciphertext to decrypt
- **msgSz** size of the buffer to decrypt
- **out** pointer to the buffer in which to store the decrypted plaintext
- **outSz** pointer to a word32 object containing the available size in the out buffer. Upon successfully decrypting the ciphertext, holds the number of bytes written to the output buffer
- **ctx** Optional: pointer to an ecEncCtx object specifying different decryption algorithms to use

See:

- wc_ecc_encrypt

- wc_ecc_encrypt_ex
- wc_ecc_ctx_set_dev_id

Return:

- 0 Returned upon successfully decrypting the input message
- BAD_FUNC_ARG Returned if privKey, msg, msgSz, out, or outSz are NULL (or pubKey is NULL when built with WOLFSSL_ECIES_OLD), or the ctx object specifies an unsupported encryption type
- BAD_ENC_STATE_E Returned if the ctx object given is in a state that is not appropriate for decryption
- BUFFER_E Returned if the supplied output buffer is too small to store the decrypted plaintext
- MEMORY_E Returned if there is an error allocating memory for the shared secret key
- NOT_COMPILED_IN Returned if an AES-GCM DEM algorithm (ecAES_128_GCM or ecAES_256_GCM) is requested in the default IV mode without the WOLFSSL_ECIES_STATIC_GCM_NONCE build macro defined

Note:

- Selecting an AES-GCM DEM algorithm (ecAES_128_GCM, ecAES_256_GCM) in the default IV mode requires the WOLFSSL_ECIES_STATIC_GCM_NONCE build macro; otherwise this function returns NOT_COMPILED_IN. See wc_ecc_encrypt_ex for the full rationale.
- The device this runs on comes from the context (wc_ecc_ctx_set_dev_id), not from privKey->devId. A NULL context, or one that was never given a device, runs in software.

Example

```
byte cipher[] = { initialize with
ciphertext to decrypt. Ensure padded to block size };
byte plain[sizeof(cipher)];
word32 plainSz = sizeof(plain);
int ret;
ecc_key cli, serv;
// initialize cli with private key
// initialize serv with received public key
ecEncCtx* cliCtx, servCtx;
// initialize cliCtx and servCtx
// exchange salts
ret = wc_ecc_decrypt(&cli, &serv, cipher, sizeof(cipher),
plain, &plainSz, cliCtx);

if(ret != 0) {
    // error decrypting message
}
```

```
int wc_ecc_set_nonblock(
    ecc_key * key,
    ecc_nb_ctx_t * ctx
)
```

Enable ECC support for non-blocking operations. Supported for Single Precision (SP) math with the following build options: WOLFSSL_SP_NONBLOCK WOLFSSL_SP_SMALL WOLFSSL_SP_NO_MALLOC WC_ECC_NONBLOCK.

Parameters:

- **key** pointer to the ecc_key object
- **ctx** pointer to ecc nb_ctx_t structure with stack data cache for SP

Return: 0 Returned upon successfully setting the callback context the input message

Example

```
int ret;
ecc_key ecc;
ecc_nb_ctx_t nb_ctx;

ret = wc_ecc_init(&ecc);
if (ret == 0) {
    ret = wc_ecc_set_nonblock(&ecc, &nb_ctx);
    if (ret == 0) {
        do {
            ret = wc_ecc_verify_hash_ex(
                &r, &s,       // r/s as mp_int
                hash, hashSz, // computed hash digest
                &verify_res,  // verification result 1=success
                &key
            );

            // TODO: Real-time work can be called here
        } while (ret == FP_WOULDBLOCK);
    }
    wc_ecc_free(&key);
}
```

```
int wc_ecc_set_curve(
    ecc_key * key,
    int keysize,
    int curve_id
)
```

Compare a curve which has larger key than specified size or the curve matched curve ID, set a curve with smaller key size to the key.

Parameters:

- **keysize** Key size in bytes
- **curve_id** Curve ID

```
 int ret;
ecc_key ecc;

ret = wc_ecc_init(&ecc);
if (ret != 0)
    return ret;
    ret = wc_ecc_set_curve(&ecc, 32, ECC_SECP256R1));
    if (ret != 0)
        return ret;
```

Return: 0 Returned upon successfully setting the key

```
mp_int * wc_ecc_key_get_priv(
    ecc_key * key
)
```

Gets private key mp_int from ECC key.

Parameters:

- **key** ECC key structure

See: wc_ecc_init

Return:

- mp_int pointer on success
- NULL on failure

Example

```
ecc_key key;
mp_int* priv = wc_ecc_key_get_priv(&key);
```

```
size_t wc_ecc_get_sets_count(
    void
)
```

Returns number of supported ECC curve sets.

See: wc_ecc_get_curve_params

Return: Number of curve sets

Example

```
size_t count = wc_ecc_get_sets_count();
```

```
const char * wc_ecc_get_name(
    int curve_id
)
```

Gets curve name from curve ID.

Parameters:

- **curve_id** Curve identifier

See: wc_ecc_get_curve_id

Return:

- Curve name string on success
- NULL on failure

Example

```
const char* name = wc_ecc_get_name(ECC_SECP256R1);
```

```
int wc_ecc_make_key_ex2(
    WC_RNG * rng,
    int keysize,
    ecc_key * key,
    int curve_id,
    int flags
)
```

Makes ECC key with extended options.

Parameters:

- **rng** Random number generator
- **keysize** Key size in bytes
- **key** ECC key structure
- **curve_id** Curve identifier
- **flags** Additional flags

See: wc_ecc_make_key_ex

Return:

- 0 on success
- negative on error

Example

```
WC_RNG rng;
ecc_key key;
int ret = wc_ecc_make_key_ex2(&rng, 32, &key,
                              ECC_SECP256R1, 0);
```

```
int wc_ecc_is_point(
    ecc_point * ecp,
    mp_int * a,
    mp_int * b,
    mp_int * prime
)
```

Checks if point is on curve.

Parameters:

- **ecp** ECC point
- **a** Curve parameter a
- **b** Curve parameter b
- **prime** Curve prime

See: wc_ecc_point_is_on_curve

Return:

- 1 if point is on curve
- 0 if not on curve
- negative on error

Example

```
ecc_point* point;
mp_int a, b, prime;
int ret = wc_ecc_is_point(point, &a, &b, &prime);
```

```
int wc_ecc_get_generator(
    ecc_point * ecp,
    int curve_idx
)
```

Gets generator point for curve.

Parameters:

- **ecp** ECC point to store generator
- **curve_idx** Curve index

See: wc_ecc_get_curve_params

Return:

- 0 on success
- negative on error

Example

```
ecc_point* gen = wc_ecc_new_point();
int ret = wc_ecc_get_generator(gen, 0);
```

```
int wc_ecc_set_deterministic(
    ecc_key * key,
    byte flag
)
```

Sets deterministic signing mode.

Parameters:

- **key** ECC key
- **flag** Enable/disable flag

See: wc_ecc_set_deterministic_ex

Return:

- 0 on success
- negative on error

Example

```
ecc_key key;
int ret = wc_ecc_set_deterministic(&key, 1);
```

```
int wc_ecc_set_deterministic_ex(
    ecc_key * key,
    byte flag,
    enum wc_HashType hashType
)
```

Sets deterministic signing with hash type.

Parameters:

- **key** ECC key
- **flag** Enable/disable flag

- **hashType** Hash algorithm type

See: wc_ecc_set_deterministic

Return:

- 0 on success
- negative on error

Example

```
ecc_key key;
int ret = wc_ecc_set_deterministic_ex(&key, 1, WC_HASH_TYPE_SHA256);
```

```
int wc_ecc_gen_deterministic_k(
    const byte * hash,
    word32 hashSz,
    enum wc_HashType hashType,
    mp_int * priv,
    mp_int * k,
    mp_int * order,
    void * heap
)
```

Generates deterministic k value for signing.

Parameters:

- **hash** Hash value
- **hashSz** Hash size
- **hashType** Hash algorithm type
- **priv** Private key
- **k** Output k value
- **order** Curve order
- **heap** Heap hint

See: wc_ecc_sign_set_k

Return:

- 0 on success
- negative on error

Example

```
byte hash[32];
mp_int priv, k, order;
int ret = wc_ecc_gen_deterministic_k(hash, 32,
                                     WC_HASH_TYPE_SHA256,
                                     &priv, &k, &order, NULL);
```

```
int wc_ecc_sign_set_k(
    const byte * k,
    word32 klen,
    ecc_key * key
)
```

Sets k value for signing.

Parameters:

- **k** K value buffer
- **klen** K value length
- **key** ECC key

See: wc_ecc_gen_deterministic_k

Return:

- 0 on success
- negative on error

Example

```
byte k[32];
ecc_key key;
int ret = wc_ecc_sign_set_k(k, sizeof(k), &key);
```

```
int wc_ecc_init_id(
    ecc_key * key,
    unsigned char * id,
    int len,
    void * heap,
    int devId
)
```

Initializes ECC key with ID.

Parameters:

- **key** ECC key
- **id** ID buffer
- **len** ID length
- **heap** Heap hint
- **devId** Device ID

See: wc_ecc_init_label

Return:

- 0 on success
- negative on error

Note: This API is only available when WOLF_PRIVATE_KEY_ID is defined, which is set for PKCS11 support.

Example

```
ecc_key key;
unsigned char id[] = "mykey";
int ret = wc_ecc_init_id(&key, id, sizeof(id), NULL,
                         INVALID_DEVID);
```

```
int wc_ecc_init_label(
    ecc_key * key,
    const char * label,
    void * heap,
    int devId
)
```

Initializes ECC key with label.

Parameters:

- **key** ECC key
- **label** Label string
- **heap** Heap hint
- **devId** Device ID

See: wc_ecc_init_id

Return:

- 0 on success
- negative on error

Note: This API is only available when WOLF_PRIVATE_KEY_ID is defined, which is set for PKCS11 support.

Example

```
ecc_key key;
int ret = wc_ecc_init_label(&key, "mykey", NULL,
                            INVALID_DEVID);
```

```
int wc_ecc_set_flags(
    ecc_key * key,
    word32 flags
)
```

Sets flags on ECC key.

Parameters:

- **key** ECC key
- **flags** Flags to set

See: wc_ecc_init

Return:

- 0 on success
- negative on error

Example

```
ecc_key key;
int ret = wc_ecc_set_flags(&key, WC_ECC_FLAG_COFACTOR);
```

```
void wc_ecc_fp_init(
    void
)
```

Initializes fixed-point cache.

See: wc_ecc_init

Return: none No returns

Example

```
wc_ecc_fp_init();
```

```
int wc_ecc_set_rng(
    ecc_key * key,
    WC_RNG * rng
)
```

Sets RNG for ECC key.

Parameters:

- **key** ECC key
- **rng** Random number generator

See: wc_ecc_make_key

Return:

- 0 on success
- negative on error

Example

```
ecc_key key;
WC_RNG rng;
int ret = wc_ecc_set_rng(&key, &rng);
```

```
int wc_ecc_get_curve_idx(
    int curve_id
)
```

Gets curve index from curve ID.

Parameters:

- **curve_id** Curve identifier

See: wc_ecc_get_curve_id

Return:

- Curve index on success
- negative on error

Example

```
int idx = wc_ecc_get_curve_idx(ECC_SECP256R1);
```

```
int wc_ecc_get_curve_id(
    int curve_idx
)
```

Gets curve ID from curve index.

Parameters:

- **curve_idx** Curve index

See: wc_ecc_get_curve_idx

Return:

- Curve ID on success
- negative on error

Example

```
int id = wc_ecc_get_curve_id(0);
```

```
int wc_ecc_get_curve_size_from_id(
    int curve_id
)
```

Gets curve size from curve ID.

Parameters:

- **curve_id** Curve identifier

See: wc_ecc_get_curve_id

Return:

- Key size in bytes on success
- negative on error

Example

```
int size = wc_ecc_get_curve_size_from_id(ECC_SECP256R1);
```

```
int wc_ecc_get_curve_idx_from_name(
    const char * curveName
)
```

Gets curve index from curve name.

Parameters:

- **curveName** Curve name string

See: wc_ecc_get_name

Return:

- Curve index on success
- negative on error

Example

```
int idx = wc_ecc_get_curve_idx_from_name("SECP256R1");
```

```
int wc_ecc_get_curve_size_from_name(
    const char * curveName
)
```

Gets curve size from curve name.

Parameters:

- **curveName** Curve name string

See: wc_ecc_get_curve_idx_from_name

Return:

- Key size in bytes on success
- negative on error

Example

```
int size = wc_ecc_get_curve_size_from_name("SECP256R1");
```

```
int wc_ecc_get_curve_id_from_name(
    const char * curveName
)
```

Gets curve ID from curve name.

Parameters:

- **curveName** Curve name string

See: wc_ecc_get_name

Return:

- Curve ID on success
- negative on error

Example

```
int id = wc_ecc_get_curve_id_from_name("SECP256R1");
```

```
int wc_ecc_get_curve_id_from_params(
    int fieldSize,
    const byte * prime,
    word32 primeSz,
    const byte * Af,
    word32 AfSz,
    const byte * Bf,
    word32 BfSz,
    const byte * order,
    word32 orderSz,
```

```
    const byte * Gx,
    word32 GxSz,
    const byte * Gy,
    word32 GySz,
    int cofactor
)
```

Gets curve ID from curve parameters.

Parameters:

- **fieldSize** Field size
- **prime** Prime modulus
- **primeSz** Prime size
- **Af** Curve parameter A
- **AfSz** A size
- **Bf** Curve parameter B
- **BfSz** B size
- **order** Curve order
- **orderSz** Order size
- **Gx** Generator X coordinate
- **GxSz** Gx size
- **Gy** Generator Y coordinate
- **GySz** Gy size
- **cofactor** Curve cofactor

See: wc_ecc_get_curve_params

Return:

- Curve ID on success
- negative on error

Example

```
int id = wc_ecc_get_curve_id_from_params(256, prime, 32,
                                         Af, 32, Bf, 32,
                                         order, 32, Gx, 32,
                                         Gy, 32, 1);
```

```
int wc_ecc_get_curve_id_from_dp_params(
    const ecc_set_type * dp
)
```

Gets curve ID from domain parameters.

Parameters:

- **dp** Domain parameters

See: wc_ecc_get_curve_params

Return:

- Curve ID on success
- negative on error

Example

```
const ecc_set_type* dp;
int id = wc_ecc_get_curve_id_from_dp_params(dp);
```

```
int wc_ecc_get_curve_id_from_oid(
    const byte * oid,
    word32 len
)
```

Gets curve ID from OID.

Parameters:

- **oid** OID buffer
- **len** OID length

See: wc_ecc_get_oid

Return:

- Curve ID on success
- negative on error

Example

```
byte oid[] = {0x2A, 0x86, 0x48, 0xCE, 0x3D, 0x03, 0x01, 0x07};
int id = wc_ecc_get_curve_id_from_oid(oid, sizeof(oid));
```

```
const ecc_set_type * wc_ecc_get_curve_params(
    int curve_idx
)
```

Gets curve parameters from curve index.

Parameters:

- **curve_idx** Curve index

See: wc_ecc_get_curve_idx

Return:

- ecc_set_type pointer on success
- NULL on failure

Example

```
const ecc_set_type* params = wc_ecc_get_curve_params(0);
```

```
ecc_point * wc_ecc_new_point_h(
    void * h
)
```

Allocates new ECC point with heap hint.

Parameters:

- **h** Heap hint

See: wc_ecc_del_point_h

Return:

- ecc_point pointer on success
- NULL on failure

Example

```
ecc_point* point = wc_ecc_new_point_h(NULL);
if (point != NULL) {
    // use point
    wc_ecc_del_point_h(point, NULL);
}
```

```
void wc_ecc_del_point_h(
    ecc_point * p,
    void * h
)
```

Frees ECC point with heap hint.

Parameters:

- **p** ECC point to free
- **h** Heap hint

See: wc_ecc_new_point_h

Return: none No returns

Example

```
ecc_point* point = wc_ecc_new_point_h(NULL);
// use point
wc_ecc_del_point_h(point, NULL);
```

```
void wc_ecc_forcezero_point(
    ecc_point * p
)
```

Securely zeros ECC point.

Parameters:

- **p** ECC point to zero

See: wc_ecc_del_point

Return: none No returns

Example

```
ecc_point* point;
wc_ecc_forcezero_point(point);
```

```
int wc_ecc_point_is_on_curve(
    ecc_point * p,
    int curve_idx
)
```

Checks if point is on curve.

Parameters:

- **p** ECC point
- **curve_idx** Curve index

See: wc_ecc_is_point

Return:

- 1 if on curve
- 0 if not on curve
- negative on error

Example

```
ecc_point* point;
int ret = wc_ecc_point_is_on_curve(point, 0);
```

```
int wc_ecc_import_x963_ex(
    const byte * in,
    word32 inLen,
    ecc_key * key,
    int curve_id
)
```

Imports X9.63 format with curve ID.

Parameters:

- **in** Input buffer
- **inLen** Input length
- **key** ECC key
- **curve_id** Curve identifier

See: wc_ecc_import_x963

Return:

- 0 on success
- negative on error

Example

```
byte x963[65];
ecc_key key;
int ret = wc_ecc_import_x963_ex(x963, sizeof(x963), &key,
                                ECC_SECP256R1);
```

```
int wc_ecc_import_private_key_ex(
    const byte * priv,
    word32 privSz,
    const byte * pub,
    word32 pubSz,
    ecc_key * key,
    int curve_id
)
```

Imports private key with curve ID.

Parameters:

- **priv** Private key buffer
- **privSz** Private key size
- **pub** Public key buffer
- **pubSz** Public key size
- **key** ECC key
- **curve_id** Curve identifier

See: wc_ecc_import_private_key

Return:

- 0 on success
- negative on error

Example

```
byte priv[32], pub[65];
ecc_key key;
int ret = wc_ecc_import_private_key_ex(priv, 32, pub, 65,
                                       &key, ECC_SECP256R1);
```

```
int wc_ecc_rs_raw_to_sig(
    const byte * r,
    word32 rSz,
    const byte * s,
    word32 sSz,
    byte * out,
    word32 * outlen
)
```

Converts raw r,s to signature.

Parameters:

- **r** R value buffer
- **rSz** R value size
- **s** S value buffer
- **sSz** S value size
- **out** Output signature buffer
- **outlen** Output signature length

See: wc_ecc_sig_to_rs

Return:

- 0 on success
- negative on error

Example

```
byte r[32], s[32], sig[72];
word32 sigLen = sizeof(sig);
int ret = wc_ecc_rs_raw_to_sig(r, 32, s, 32, sig, &sigLen);
```

```
int wc_ecc_sig_to_rs(
    const byte * sig,
    word32 sigLen,
    byte * r,
    word32 * rLen,
    byte * s,
    word32 * sLen
)
```

Converts signature to raw r,s.

Parameters:

- **sig** Signature buffer
- **sigLen** Signature length
- **r** R value buffer
- **rLen** R value length
- **s** S value buffer
- **sLen** S value length

See: wc_ecc_rs_raw_to_sig

Return:

- 0 on success
- negative on error

Example

```
byte sig[72], r[32], s[32];
word32 rLen = 32, sLen = 32;
int ret = wc_ecc_sig_to_rs(sig, 72, r, &rLen, s, &sLen);
```

```
int wc_ecc_import_raw_ex(
    ecc_key * key,
    const char * qx,
    const char * qy,
    const char * d,
    int curve_id
)
```

Imports raw key with curve ID.

Parameters:

- **key** ECC key
- **qx** X coordinate string
- **qy** Y coordinate string
- **d** Private key string
- **curve_id** Curve identifier

See: wc_ecc_import_raw

Return:

- 0 on success
- negative on error

Note: This function does not check that the imported public point lies on the curve. Define WOLFSSL_VALIDATE_ECC_IMPORT to validate the point on import, or call wc_ecc_check_key before the key is used.

Example

```
ecc_key key;
int ret = wc_ecc_import_raw_ex(&key, qxStr, qyStr, dStr,
                               ECC_SECP256R1);
```

```
int wc_ecc_import_unsigned(
    ecc_key * key,
    const byte * qx,
    const byte * qy,
    const byte * d,
    int curve_id
)
```

Imports unsigned key with curve ID.

Parameters:

- **key** ECC key
- **qx** X coordinate buffer
- **qy** Y coordinate buffer
- **d** Private key buffer
- **curve_id** Curve identifier

See: wc_ecc_import_raw_ex

Return:

- 0 on success
- negative on error

Note: This function does not check that the imported public point lies on the curve. Define WOLFSSL_VALIDATE_ECC_IMPORT to validate the point on import, or call wc_ecc_check_key before the key is used.

Example

```
ecc_key key;
byte qx[32], qy[32], d[32];
int ret = wc_ecc_import_unsigned(&key, qx, qy, d,
                                 ECC_SECP256R1);
```

```
int wc_ecc_export_ex(
    ecc_key * key,
    byte * qx,
    word32 * qxLen,
    byte * qy,
    word32 * qyLen,
    byte * d,
    word32 * dLen,
    int encType
)
```

Exports key with encoding type.

Parameters:

- **key** ECC key
- **qx** X coordinate buffer
- **qxLen** X coordinate length
- **qy** Y coordinate buffer
- **qyLen** Y coordinate length
- **d** Private key buffer
- **dLen** Private key length
- **encType** Encoding type

See: wc_ecc_export_public_raw

Return:

- 0 on success
- negative on error

Example

```
ecc_key key;
byte qx[32], qy[32], d[32];
word32 qxLen = 32, qyLen = 32, dLen = 32;
int ret = wc_ecc_export_ex(&key, qx, &qxLen, qy, &qyLen,
                           d, &dLen, 0);
```

```
int wc_ecc_export_public_raw(
    ecc_key * key,
    byte * qx,
    word32 * qxLen,
```

```
    byte * qy,
    word32 * qyLen
)
```

Exports public key in raw format.

Parameters:

- **key** ECC key
- **qx** X coordinate buffer
- **qxLen** X coordinate length
- **qy** Y coordinate buffer
- **qyLen** Y coordinate length

See: wc_ecc_export_private_raw

Return:

- 0 on success
- negative on error

Example

```
ecc_key key;
byte qx[32], qy[32];
word32 qxLen = 32, qyLen = 32;
int ret = wc_ecc_export_public_raw(&key, qx, &qxLen, qy,
                                   &qyLen);
```

```
int wc_ecc_export_private_raw(
    ecc_key * key,
    byte * qx,
    word32 * qxLen,
    byte * qy,
    word32 * qyLen,
    byte * d,
    word32 * dLen
)
```

Exports private key in raw format.

Parameters:

- **key** ECC key
- **qx** X coordinate buffer
- **qxLen** X coordinate length
- **qy** Y coordinate buffer
- **qyLen** Y coordinate length
- **d** Private key buffer
- **dLen** Private key length

See: wc_ecc_export_public_raw

Return:

- 0 on success
- negative on error

Example

```
ecc_key key;
byte qx[32], qy[32], d[32];
word32 qxLen = 32, qyLen = 32, dLen = 32;
int ret = wc_ecc_export_private_raw(&key, qx, &qxLen, qy,
                                    &qyLen, d, &dLen);
```

```
int wc_ecc_export_point_der_ex(
    const int curve_idx,
    ecc_point * point,
    byte * out,
    word32 * outLen,
    int compressed
)
```

Exports point in DER format with compression.

Parameters:

- **curve_idx** Curve index
- **point** ECC point
- **out** Output buffer
- **outLen** Output length
- **compressed** Compression flag

See: wc_ecc_export_point_der

Return:

- Size on success
- negative on error

Example

```
ecc_point* point;
byte out[65];
word32 outLen = sizeof(out);
int ret = wc_ecc_export_point_der_ex(0, point, out, &outLen,
                                     0);
```

```
int wc_ecc_import_point_der_ex(
    const byte * in,
    word32 inLen,
    const int curve_idx,
    ecc_point * point,
    int shortKeySize
)
```

Imports point from DER format.

Parameters:

- **in** Input buffer
- **inLen** Input length
- **curve_idx** Curve index
- **point** ECC point
- **shortKeySize** Short key size flag

See: wc_ecc_import_point_der

Return:

- 0 on success
- negative on error

Example

```
byte der[65];
ecc_point* point = wc_ecc_new_point();
int ret = wc_ecc_import_point_der_ex(der, sizeof(der), 0,
                                     point, 0);
```

```
int wc_ecc_get_oid(
    word32 oidSum,
    const byte ** oid,
    word32 * oidSz
)
```

Gets OID for curve.

Parameters:

- **oidSum** OID sum
- **oid** OID buffer pointer
- **oidSz** OID size pointer

See: wc_ecc_get_curve_id_from_oid

Return:

- 0 on success

- negative on error

Example

```
const byte* oid;
word32 oidSz;
int ret = wc_ecc_get_oid(0x2A8648CE3D030107, &oid, &oidSz);
```

```
int wc_ecc_set_custom_curve(
    ecc_key * key,
    const ecc_set_type * dp
)
```

Sets custom curve parameters.

Parameters:

- **key** ECC key
- **dp** Domain parameters

See: wc_ecc_get_curve_params

Return:

- 0 on success
- negative on error

Example

```
ecc_key key;
ecc_set_type dp;
int ret = wc_ecc_set_custom_curve(&key, &dp);
```

```
ecEncCtx * wc_ecc_ctx_new_ex(
    int flags,
    WC_RNG * rng,
    void * heap
)
```

Creates new ECC encryption context with heap.

Parameters:

- **flags** Context flags
- **rng** Random number generator
- **heap** Heap hint

See: wc_ecc_ctx_new

Return:

- ecEncCtx pointer on success
- NULL on failure

Example

```
WC_RNG rng;
ecEncCtx* ctx = wc_ecc_ctx_new_ex(0, &rng, NULL);
```

```
int wc_ecc_ctx_set_own_salt(
    ecEncCtx * ctx,
    const byte * salt,
    word32 sz
)
```

Sets own salt in context.

Parameters:

- **ctx** ECC encryption context
- **salt** Salt buffer
- **sz** Salt size

See: wc_ecc_ctx_get_own_salt

Return:

- 0 on success
- negative on error

Example

```
ecEncCtx* ctx;
byte salt[16];
int ret = wc_ecc_ctx_set_own_salt(ctx, salt, sizeof(salt));
```

```
int wc_X963_KDF(
    enum wc_HashType type,
    const byte * secret,
    word32 secretSz,
    const byte * sinfo,
    word32 sinfoSz,
    byte * out,
    word32 outSz
)
```

X9.63 Key Derivation Function.

Parameters:

- **type** Hash type
- **secret** Shared secret
- **secretSz** Secret size
- **sinfo** Shared info
- **sinfoSz** Shared info size
- **out** Output buffer
- **outSz** Output size

See: wc_ecc_shared_secret

Return:

- 0 on success
- negative on error

Example

```
byte secret[32], sinfo[10], out[32];
int ret = wc_X963_KDF(WC_HASH_TYPE_SHA256, secret, 32,
                      sinfo, 10, out, 32);
```

```
int wc_ecc_curve_cache_init(
    void
)
```

Initializes curve cache.

See: wc_ecc_curve_cache_free

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_ecc_curve_cache_init();
```

```
void wc_ecc_curve_cache_free(
    void
)
```

Frees curve cache.

See: wc_ecc_curve_cache_init

Return: none No returns

Example

```
wc_ecc_curve_cache_free();
```

```
int wc_ecc_gen_k(
    WC_RNG * rng,
    int size,
    mp_int * k,
    mp_int * order
)
```

Generates random k value.

Parameters:

- **rng** Random number generator
- **size** Key size
- **k** Output k value
- **order** Curve order

See: wc_ecc_sign_hash

Return:

- 0 on success
- negative on error

Example

```
WC_RNG rng;
mp_int k, order;
int ret = wc_ecc_gen_k(&rng, 32, &k, &order);
```

```
int wc_ecc_set_handle(
    ecc_key * key,
    remote_handle64 handle
)
```

Sets remote handle for hardware.

Parameters:

- **key** ECC key
- **handle** Remote handle

See: wc_ecc_init

Return:

- 0 on success
- negative on error

Example

```
ecc_key key;
remote_handle64 handle = 0x1234;
int ret = wc_ecc_set_handle(&key, handle);
```

```
int wc_ecc_use_key_id(
    ecc_key * key,
    word32 keyId,
    word32 flags
)
```

Uses key ID for hardware.

Parameters:

- **key** ECC key
- **keyId** Key identifier
- **flags** Flags

See: wc_ecc_get_key_id

Return:

- 0 on success
- negative on error

Example

```
ecc_key key;
int ret = wc_ecc_use_key_id(&key, 1, 0);
```

```
int wc_ecc_get_key_id(
    ecc_key * key,
    word32 * keyId
)
```

Gets key ID from hardware key.

Parameters:

- **key** ECC key
- **keyId** Key identifier pointer

See: wc_ecc_use_key_id

Return:

- 0 on success
- negative on error

Example

```
ecc_key key;
word32 keyId;
int ret = wc_ecc_get_key_id(&key, &keyId);
```

B.25 Algorithms - ED25519

B.24.2.110 function wc_ecc_get_key_id

B.25.1 Functions

	Name
int	**wc_ed25519_make_public**(ed25519_key * key, unsigned char * pubKey, word32 pubKeySz)This function generates the Ed25519 public key from the private key, stored in the ed25519_key object. It stores the public key in the buffer pubKey. If the key object does not already carry a public key, the derived key is also stored in it, so that the key can be used for signing.
int	**wc_ed25519_make_key**(WC_RNG * rng, int keysize, ed25519_key * key)This function generates a new Ed25519 key and stores it in key.
int	**wc_ed25519_sign_msg**(const byte * in, word32 inlen, byte * out, word32 * outlen, ed25519_key * key)This function signs a message using an ed25519_key object to guarantee authenticity.
int	**wc_ed25519ctx_sign_msg**(const byte * in, word32 inlen, byte * out, word32 * outlen, ed25519_key * key, const byte * context, byte contextLen)This function signs a message using an ed25519_key object to guarantee authenticity. The context is part of the data signed.

	Name
int	**wc_ed25519ph_sign_hash**(const byte * hash, word32 hashLen, byte * out, word32 * outLen, ed25519_key * key, const byte * context, byte contextLen)This function signs a message digest using an ed25519_key object to guarantee authenticity. The context is included as part of the data signed. The message is pre-hashed before signature calculation.
int	**wc_ed25519ph_sign_msg**(const byte * in, word32 inlen, byte * out, word32 * outlen, ed25519_key * key, const byte * context, byte contextLen)This function signs a message using an ed25519_key object to guarantee authenticity. The context is included as part of the data signed. The message is pre-hashed before signature calculation.
int	**wc_ed25519_verify_msg**(const byte * sig, word32 siglen, const byte * msg, word32 msgLen, int * ret, ed25519_key * key)This function verifies the Ed25519 signature of a message to ensure authenticity. It returns the answer through ret, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.
int	**wc_ed25519ctx_verify_msg**(const byte * sig, word32 siglen, const byte * msg, word32 msgLen, int * ret, ed25519_key * key, const byte * context, byte contextLen)This function verifies the Ed25519 signature of a message to ensure authenticity. The context is included as part of the data verified. It returns the answer through ret, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.
int	**wc_ed25519ph_verify_hash**(const byte * sig, word32 siglen, const byte * hash, word32 hashLen, int * ret, ed25519_key * key, const byte * context, byte contextLen)This function verifies the Ed25519 signature of the digest of a message to ensure authenticity. The context is included as part of the data verified. The hash is the pre-hashed message before signature calculation. The hash algorithm used to create message digest must be SHA-512. The answer is returned through ret, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

	Name
int	**wc_ed25519ph_verify_msg**(const byte * sig, word32 siglen, const byte * msg, word32 msgLen, int * ret, ed25519_key * key, const byte * context, byte contextLen)This function verifies the Ed25519 signature of a message to ensure authenticity. The context is included as part of the data verified. The message is pre-hashed before verification. It returns the answer through ret, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.
int	**wc_ed25519_init**(ed25519_key * key)This function initializes an ed25519_key object for future use with message verification.
void	**wc_ed25519_free**(ed25519_key * key)This function frees an Ed25519 object after it has been used.
int	**wc_ed25519_import_public**(const byte * in, word32 inLen, ed25519_key * key)This function imports a public ed25519_key from a buffer containing the public key. This function will handle both compressed and uncompressed keys. The public key is checked that it matches the private key when one is present.
int	**wc_ed25519_import_public_ex**(const byte * in, word32 inLen, ed25519_key * key, int trusted)This function imports a public ed25519_key from a buffer containing the public key. This function will handle both compressed and uncompressed keys. Check public key matches private key, when present, when not trusted.
int	**wc_ed25519_import_private_only**(const byte * priv, word32 privSz, ed25519_key * key)This function imports an Ed25519 private key only from a buffer.
int	**wc_ed25519_import_private_key**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ed25519_key * key)This function imports a public/private Ed25519 key pair from a pair of buffers. This function will handle both compressed and uncompressed keys. The public key is assumed to be untrusted and is checked against the private key.
int	**wc_ed25519_import_private_key_ex**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ed25519_key * key, int trusted)This function imports a public/private Ed25519 key pair from a pair of buffers. This function will handle both compressed and uncompressed keys. The public is checked against private key if not trusted.

	Name
int	**wc_ed25519_export_public**(const ed25519_key * key, byte * out, word32 * outLen)This function exports the public key from an ed25519_key structure. It stores the public key in the buffer out, and sets the bytes written to this buffer in outLen.
int	**wc_ed25519_export_private_only**(const ed25519_key * key, byte * out, word32 * outLen)This function exports only the private key from an ed25519_key structure. It stores the private key in the buffer out, and sets the bytes written to this buffer in outLen.
int	**wc_ed25519_export_private**(const ed25519_key * key, byte * out, word32 * outLen)This function exports the key pair from an ed25519_key structure. It stores the key pair in the buffer out, and sets the bytes written to this buffer in outLen.
int	**wc_ed25519_export_key**(const ed25519_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz)This function exports the private and public key separately from an ed25519_key structure. It stores the private key in the buffer priv, and sets the bytes written to this buffer in privSz. It stores the public key in the buffer pub, and sets the bytes written to this buffer in pubSz.
int	**wc_ed25519_check_key**(ed25519_key * key)This function checks the public key in ed25519_key structure matches the private key.
int	**wc_ed25519_size**(const ed25519_key * key)This function returns the size of an Ed25519 - 32 bytes.
int	**wc_ed25519_priv_size**(const ed25519_key * key)This function returns the private key size (secret + public) in bytes.
int	**wc_ed25519_pub_size**(const ed25519_key * key)This function returns the compressed key size in bytes (public key).
int	**wc_ed25519_sig_size**(const ed25519_key * key)This function returns the size of an Ed25519 signature (64 in bytes).
int	**wc_ed25519_sign_msg_ex**(const byte * in, word32 inLen, byte * out, word32 * outLen, ed25519_key * key, byte type, const byte * context, byte contextLen)Signs message with extended parameters.
int	**wc_ed25519_verify_msg_ex**(const byte * sig, word32 sigLen, const byte * msg, word32 msgLen, int * res, ed25519_key * key, byte type, const byte * context, byte contextLen)Verifies signature with extended parameters.

	Name
int	**wc_ed25519_verify_msg_init**(const byte * sig, word32 sigLen, ed25519_key * key, byte type, const byte * context, byte contextLen)Initializes streaming verification.
int	**wc_ed25519_verify_msg_update**(const byte * msgSegment, word32 msgSegmentLen, ed25519_key * key)Updates streaming verification with message segment.
int	**wc_ed25519_verify_msg_final**(const byte * sig, word32 sigLen, int * res, ed25519_key * key)Finalizes streaming verification.
int	**wc_ed25519_init_ex**(ed25519_key * key, void * heap, int devId)Initializes Ed25519 key with extended parameters.
int	**wc_ed25519_init_id**(ed25519_key * key, const unsigned char * id, int len, void * heap, int devId)Initializes an ed25519_key object that references a private key held by a crypto callback device, identified by id. Only available when built with WOLF_PRIVATE_KEY_ID.
int	**wc_ed25519_init_label**(ed25519_key * key, const char * label, void * heap, int devId)Initializes an ed25519_key object that references a private key held by a crypto callback device, identified by label. Only available when built with WOLF_PRIVATE_KEY_ID.
ed25519_key *	**wc_ed25519_new**(void * heap, int devId, int * result_code)Allocates and initializes new Ed25519 key. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.
int	**wc_ed25519_delete**(ed25519_key * key, ed25519_key ** key_p)Frees and deletes Ed25519 key. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.

B.25.2 Functions Documentation

```
int wc_ed25519_make_public(
    ed25519_key * key,
    unsigned char * pubKey,
    word32 pubKeySz
)
```

This function generates the Ed25519 public key from the private key, stored in the ed25519_key object. It stores the public key in the buffer pubKey. If the key object does not already carry a public key, the derived key is also stored in it, so that the key can be used for signing.

Parameters:

- **key** Pointer to the ed25519_key for which to generate a key.
- **pubKey** Pointer to the buffer in which to store the public key.
- **pubKeySz** Size of the public key. Should be ED25519_PUB_KEY_SIZE.

See:

- wc_ed25519_init
- wc_ed25519_import_private_only
- wc_ed25519_make_key

Return:

- 0 Returned upon successfully making the public key.
- BAD_FUNC_ARG Returned if key or pubKey evaluate to NULL, or if the specified key size is not 32 bytes (Ed25519 has 32 byte keys).
- ECC_PRIV_KEY_E returned if the ed25519_key object does not have the private key in it.
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
int ret;

ed25519_key key;
byte priv[] = { initialize with 32 byte private key };
byte pub[32];
word32 pubSz = sizeof(pub);

wc_ed25519_init(&key);
wc_ed25519_import_private_only(priv, sizeof(priv), &key);
ret = wc_ed25519_make_public(&key, pub, pubSz);
if (ret != 0) {
    // error making public key
}
```

```
int wc_ed25519_make_key(
    WC_RNG * rng,
    int keysize,
    ed25519_key * key
)
```

This function generates a new Ed25519 key and stores it in key.

Parameters:

- **rng** Pointer to an initialized RNG object with which to generate the key.
- **keysize** Length of key to generate. Should always be 32 for Ed25519.
- **key** Pointer to the ed25519_key for which to generate a key.

See: wc_ed25519_init

Return:

- 0 Returned upon successfully making an ed25519_key.
- BAD_FUNC_ARG Returned if rng or key evaluate to NULL, or if the specified key size is not 32 bytes (Ed25519 has 32 byte keys).
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
int ret;

WC_RNG rng;
ed25519_key key;

wc_InitRng(&rng);
wc_ed25519_init(&key);
wc_ed25519_make_key(&rng, 32, &key);
if (ret != 0) {
    // error making key
}
```

```
int wc_ed25519_sign_msg(
    const byte * in,
    word32 inlen,
    byte * out,
    word32 * outlen,
    ed25519_key * key
)
```

This function signs a message using an ed25519_key object to guarantee authenticity.

Parameters:

- **in** Pointer to the buffer containing the message to sign.
- **inlen** Length of the message to sign.
- **out** Buffer in which to store the generated signature.
- **outlen** Maximum length of the output buffer. Will store the bytes written to out upon successfully generating a message signature.
- **key** Pointer to a private ed25519_key with which to generate the signature.

See:

- wc_ed25519ctx_sign_msg
- wc_ed25519ph_sign_hash
- wc_ed25519ph_sign_msg
- wc_ed25519_verify_msg

Return:

- 0 Returned upon successfully generating a signature for the message.
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the output buffer is too small to store the generated signature.
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
ed25519_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[64]; // will hold generated signature
sigSz = sizeof(sig);
byte message[] = { initialize with message };

wc_InitRng(&rng); // initialize rng
wc_ed25519_init(&key); // initialize key
wc_ed25519_make_key(&rng, 32, &key); // make public/private key pair
ret = wc_ed25519_sign_msg(message, sizeof(message), sig, &sigSz, &key);
if (ret != 0) {
    // error generating message signature
}
```

```
int wc_ed25519ctx_sign_msg(
    const byte * in,
    word32 inlen,
    byte * out,
    word32 * outlen,
    ed25519_key * key,
    const byte * context,
    byte contextLen
)
```

This function signs a message using an ed25519_key object to guarantee authenticity. The context is part of the data signed.

Parameters:

- **in** Pointer to the buffer containing the message to sign.
- **inlen** Length of the message to sign.
- **out** Buffer in which to store the generated signature.
- **outlen** Maximum length of the output buffer. Will store the bytes written to out upon successfully generating a message signature.
- **key** Pointer to a private ed25519_key with which to generate the signature.
- **context** Pointer to the buffer containing the context for which message is being signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed25519_sign_msg
- wc_ed25519ph_sign_hash
- wc_ed25519ph_sign_msg
- wc_ed25519_verify_msg

Return:

- 0 Returned upon successfully generating a signature for the message.
- BAD_FUNC_ARG Returned any of the input parameters evaluate to NULL, or if the output buffer is too small to store the generated signature.
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
ed25519_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[64]; // will hold generated signature
sigSz = sizeof(sig);
byte message[] = { initialize with message };
byte context[] = { initialize with context of signing };

wc_InitRng(&rng); // initialize rng
wc_ed25519_init(&key); // initialize key
wc_ed25519_make_key(&rng, 32, &key); // make public/private key pair
ret = wc_ed25519ctx_sign_msg(message, sizeof(message), sig, &sigSz, &key,
        context, sizeof(context));
if (ret != 0) {
    // error generating message signature
}
```

```
int wc_ed25519ph_sign_hash(
    const byte * hash,
    word32 hashLen,
    byte * out,
    word32 * outLen,
    ed25519_key * key,
    const byte * context,
    byte contextLen
)
```

This function signs a message digest using an ed25519_key object to guarantee authenticity. The context is included as part of the data signed. The message is pre-hashed before signature calculation.

Parameters:

- **hash** Pointer to the buffer containing the hash of the message to sign.
- **hashLen** Length of the hash of the message to sign.
- **out** Buffer in which to store the generated signature.

- **outLen** Maximum length of the output buffer. Will store the bytes written to out upon successfully generating a message signature.
- **key** Pointer to a private ed25519_key with which to generate the signature.
- **context** Pointer to the buffer containing the context for which message is being signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed25519_sign_msg
- wc_ed25519ctx_sign_msg
- wc_ed25519ph_sign_msg
- wc_ed25519_verify_msg

Return:

- 0 Returned upon successfully generating a signature for the message digest.
- BAD_FUNC_ARG Returned any of the input parameters evaluate to NULL, or if the output buffer is too small to store the generated signature.
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
ed25519_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[64]; // will hold generated signature
sigSz = sizeof(sig);
byte hash[] = { initialize with SHA-512 hash of message };
byte context[] = { initialize with context of signing };

wc_InitRng(&rng); // initialize rng
wc_ed25519_init(&key); // initialize key
wc_ed25519_make_key(&rng, 32, &key); // make public/private key pair
ret = wc_ed25519ph_sign_hash(hash, sizeof(hash), sig, &sigSz, &key,
        context, sizeof(context));
if (ret != 0) {
    // error generating message signature
}
```

```
int wc_ed25519ph_sign_msg(
    const byte * in,
    word32 inlen,
    byte * out,
    word32 * outlen,
    ed25519_key * key,
    const byte * context,
    byte contextLen
)
```

This function signs a message using an ed25519_key object to guarantee authenticity. The context is included as part of the data signed. The message is pre-hashed before signature calculation.

Parameters:

- **in** Pointer to the buffer containing the message to sign.
- **inlen** Length of the message to sign.
- **out** Buffer in which to store the generated signature.
- **outlen** Maximum length of the output buffer. Will store the bytes written to out upon successfully generating a message signature.
- **key** Pointer to a private ed25519_key with which to generate the signature.
- **context** Pointer to the buffer containing the context for which message is being signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed25519_sign_msg
- wc_ed25519ctx_sign_msg
- wc_ed25519ph_sign_hash
- wc_ed25519_verify_msg

Return:

- 0 Returned upon successfully generating a signature for the message.
- BAD_FUNC_ARG Returned any of the input parameters evaluate to NULL, or if the output buffer is too small to store the generated signature.
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
ed25519_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[64]; // will hold generated signature
sigSz = sizeof(sig);
byte message[] = { initialize with message };
byte context[] = { initialize with context of signing };

wc_InitRng(&rng); // initialize rng
wc_ed25519_init(&key); // initialize key
wc_ed25519_make_key(&rng, 32, &key); // make public/private key pair
ret = wc_ed25519ph_sign_msg(message, sizeof(message), sig, &sigSz, &key,
        context, sizeof(context));
if (ret != 0) {
    // error generating message signature
}
```

```
int wc_ed25519_verify_msg(
    const byte * sig,
```

```
    word32 siglen,
    const byte * msg,
    word32 msgLen,
    int * ret,
    ed25519_key * key
)
```

This function verifies the Ed25519 signature of a message to ensure authenticity. It returns the answer through ret, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

Parameters:

- **sig** Pointer to the buffer containing the signature to verify.
- **siglen** Length of the signature to verify.
- **msg** Pointer to the buffer containing the message to verify.
- **msgLen** Length of the message to verify.
- **ret** Pointer to the result of the verification. 1 indicates the message was successfully verified.
- **key** Pointer to a public Ed25519 key with which to verify the signature.

See:

- wc_ed25519ctx_verify_msg
- wc_ed25519ph_verify_hash
- wc_ed25519ph_verify_msg
- wc_ed25519_sign_msg

Return:

- 0 Returned upon successfully performing the signature verification and authentication.
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the siglen does not match the actual length of a signature.
- SIG_VERIFY_E Returned if verification completes, but the signature generated does not match the signature provided.

Example

```
ed25519_key key;
int ret, verified = 0;

byte sig[] { initialize with received signature };
byte msg[] = { initialize with message };
// initialize key with received public key
ret = wc_ed25519_verify_msg(sig, sizeof(sig), msg, sizeof(msg), &verified,
        &key);
if (ret < 0) {
    // error performing verification
} else if (verified == 0)
    // the signature is invalid
}
```

```
int wc_ed25519ctx_verify_msg(
    const byte * sig,
    word32 siglen,
    const byte * msg,
    word32 msgLen,
    int * ret,
    ed25519_key * key,
    const byte * context,
    byte contextLen
)
```

This function verifies the Ed25519 signature of a message to ensure authenticity. The context is included as part of the data verified. It returns the answer through ret, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

Parameters:

- **sig** Pointer to the buffer containing the signature to verify.
- **siglen** Length of the signature to verify.
- **msg** Pointer to the buffer containing the message to verify.
- **msgLen** Length of the message to verify.
- **ret** Pointer to the result of the verification. 1 indicates the message was successfully verified.
- **key** Pointer to a public Ed25519 key with which to verify the signature.
- **context** Pointer to the buffer containing the context for which the message was signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed25519_verify_msg
- wc_ed25519ph_verify_hash
- wc_ed25519ph_verify_msg
- wc_ed25519_sign_msg

Return:

- 0 Returned upon successfully performing the signature verification and authentication.
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the siglen does not match the actual length of a signature.
- SIG_VERIFY_E Returned if verification completes, but the signature generated does not match the signature provided.

Example

```
ed25519_key key;
int ret, verified = 0;

byte sig[] { initialize with received signature };
byte msg[] = { initialize with message };
byte context[] = { initialize with context of signature };
// initialize key with received public key
ret = wc_ed25519ctx_verify_msg(sig, sizeof(sig), msg, sizeof(msg),
        &verified, &key, );
```

```
if (ret < 0) {
    // error performing verification
} else if (verified == 0)
    // the signature is invalid
}
```

```
int wc_ed25519ph_verify_hash(
    const byte * sig,
    word32 siglen,
    const byte * hash,
    word32 hashLen,
    int * ret,
    ed25519_key * key,
    const byte * context,
    byte contextLen
)
```

This function verifies the Ed25519 signature of the digest of a message to ensure authenticity. The context is included as part of the data verified. The hash is the pre-hashed message before signature calculation. The hash algorithm used to create message digest must be SHA-512. The answer is returned through ret, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

Parameters:

- **sig** Pointer to the buffer containing the signature to verify.
- **siglen** Length of the signature to verify.
- **hash** Pointer to the buffer containing the hash of the message to verify.
- **hashLen** Length of the hash to verify.
- **ret** Pointer to the result of the verification. 1 indicates the message was successfully verified.
- **key** Pointer to a public Ed25519 key with which to verify the signature.
- **context** Pointer to the buffer containing the context for which the message was signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed25519_verify_msg
- wc_ed25519ctx_verify_msg
- wc_ed25519ph_verify_msg
- wc_ed25519_sign_msg

Return:

- 0 Returned upon successfully performing the signature verification and authentication.
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the siglen does not match the actual length of a signature.
- SIG_VERIFY_E Returned if verification completes, but the signature generated does not match the signature provided.

Example

```
ed25519_key key;
int ret, verified = 0;

byte sig[] { initialize with received signature };
byte hash[] = { initialize with SHA-512 hash of message };
byte context[] = { initialize with context of signature };
// initialize key with received public key
ret = wc_ed25519ph_verify_hash(sig, sizeof(sig), hash, sizeof(hash),
        &verified, &key, context, sizeof(context));
if (ret < 0) {
    // error performing verification
} else if (verified == 0)
    // the signature is invalid
}
```

```
int wc_ed25519ph_verify_msg(
    const byte * sig,
    word32 siglen,
    const byte * msg,
    word32 msgLen,
    int * ret,
    ed25519_key * key,
    const byte * context,
    byte contextLen
)
```

This function verifies the Ed25519 signature of a message to ensure authenticity. The context is included as part of the data verified. The message is pre-hashed before verification. It returns the answer through ret, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

Parameters:

- **sig** Pointer to the buffer containing the signature to verify.
- **siglen** Length of the signature to verify.
- **msg** Pointer to the buffer containing the message to verify.
- **msgLen** Length of the message to verify.
- **ret** Pointer to the result of the verification. 1 indicates the message was successfully verified.
- **key** Pointer to a public Ed25519 key with which to verify the signature.
- **context** Pointer to the buffer containing the context for which the message was signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed25519_verify_msg
- wc_ed25519ctx_verify_msg
- wc_ed25519ph_verify_hash
- wc_ed25519_sign_msg

Return:

- 0 Returned upon successfully performing the signature verification and authentication.
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the siglen does not match the actual length of a signature.
- SIG_VERIFY_E Returned if verification completes, but the signature generated does not match the signature provided.

Example

```
ed25519_key key;
int ret, verified = 0;

byte sig[] { initialize with received signature };
byte msg[] = { initialize with message };
byte context[] = { initialize with context of signature };
// initialize key with received public key
ret = wc_ed25519ph_verify_msg(sig, sizeof(sig), msg, sizeof(msg),
        &verified, &key, context, sizeof(context));
if (ret < 0) {
    // error performing verification
} else if (verified == 0)
    // the signature is invalid
}
```

```
int wc_ed25519_init(
    ed25519_key * key
)
```

This function initializes an ed25519_key object for future use with message verification.

Parameters:

- **key** Pointer to the ed25519_key object to initialize.

See:

- wc_ed25519_make_key
- wc_ed25519_free

Return:

- 0 Returned upon successfully initializing the ed25519_key object.
- BAD_FUNC_ARG Returned if key is NULL.

Example

```
ed25519_key key;
wc_ed25519_init(&key);
```

```
void wc_ed25519_free(
    ed25519_key * key
)
```

This function frees an Ed25519 object after it has been used.

Parameters:

- **key** Pointer to the ed25519_key object to free

See: wc_ed25519_init

Example

```
ed25519_key key;
// initialize key and perform secure exchanges
...
wc_ed25519_free(&key);
```

```
int wc_ed25519_import_public(
    const byte * in,
    word32 inLen,
    ed25519_key * key
)
```

This function imports a public ed25519_key from a buffer containing the public key. This function will handle both compressed and uncompressed keys. The public key is checked that it matches the private key when one is present.

Parameters:

- **in** Pointer to the buffer containing the public key.
- **inLen** Length of the buffer containing the public key.
- **key** Pointer to the ed25519_key object in which to store the public key.

See:

- wc_ed25519_import_public_ex
- wc_ed25519_import_private_key
- wc_ed25519_import_private_key_ex
- wc_ed25519_export_public

Return:

- 0 Returned on successfully importing the ed25519_key.
- BAD_FUNC_ARG Returned if in or key evaluate to NULL, or inLen is less than the size of an Ed25519 key.

Example

```
int ret;
byte pub[] = { initialize Ed25519 public key };

ed_25519 key;
wc_ed25519_init_key(&key);
ret = wc_ed25519_import_public(pub, sizeof(pub), &key);
if (ret != 0) {
    // error importing key
}
```

```
int wc_ed25519_import_public_ex(
    const byte * in,
    word32 inLen,
    ed25519_key * key,
    int trusted
)
```

This function imports a public ed25519_key from a buffer containing the public key. This function will handle both compressed and uncompressed keys. Check public key matches private key, when present, when not trusted.

Parameters:

- **in** Pointer to the buffer containing the public key.
- **inLen** Length of the buffer containing the public key.
- **key** Pointer to the ed25519_key object in which to store the public key.
- **trusted** Public key data is trusted or not.

See:

- wc_ed25519_import_public
- wc_ed25519_import_private_key
- wc_ed25519_import_private_key_ex
- wc_ed25519_export_public

Return:

- 0 Returned on successfully importing the ed25519_key.
- BAD_FUNC_ARG Returned if in or key evaluate to NULL, or inLen is less than the size of an Ed25519 key.

Example

```
int ret;
byte pub[] = { initialize Ed25519 public key };

ed_25519 key;
wc_ed25519_init_key(&key);
ret = wc_ed25519_import_public_ex(pub, sizeof(pub), &key, 1);
if (ret != 0) {
```

```
    // error importing key
}
```

```
int wc_ed25519_import_private_only(
    const byte * priv,
    word32 privSz,
    ed25519_key * key
)
```

This function imports an Ed25519 private key only from a buffer.

Parameters:

- **priv** Pointer to the buffer containing the private key.
- **privSz** Length of the private key.
- **key** Pointer to the ed25519_key object in which to store the imported private key.

See:

- wc_ed25519_import_public
- wc_ed25519_import_public_ex
- wc_ed25519_import_private_key
- wc_ed25519_import_private_key_ex
- wc_ed25519_export_private_only

Return:

- 0 Returned on successfully importing the Ed25519 key.
- BAD_FUNC_ARG Returned if priv or key evaluate to NULL, or if privSz is not equal to ED25519_KEY_SIZE.

Example

```
int ret;
byte priv[] = { initialize with 32 byte private key };

ed25519_key key;
wc_ed25519_init_key(&key);
ret = wc_ed25519_import_private_only(priv, sizeof(priv), &key);
if (ret != 0) {
    // error importing private key
}
```

```
int wc_ed25519_import_private_key(
    const byte * priv,
    word32 privSz,
    const byte * pub,
    word32 pubSz,
```

```
    ed25519_key * key
)
```

This function imports a public/private Ed25519 key pair from a pair of buffers. This function will handle both compressed and uncompressed keys. The public key is assumed to be untrusted and is checked against the private key.

Parameters:

- **priv** Pointer to the buffer containing the private key.
- **privSz** Length of the private key.
- **pub** Pointer to the buffer containing the public key.
- **pubSz** Length of the public key.
- **key** Pointer to the ed25519_key object in which to store the imported private/public key pair.

See:

- wc_ed25519_import_public
- wc_ed25519_import_public_ex
- wc_ed25519_import_private_only
- wc_ed25519_import_private_key_ex
- wc_ed25519_export_private

Return:

- 0 Returned on successfully importing the ed25519_key.
- BAD_FUNC_ARG Returned if priv or key evaluate to NULL; or if either privSz is not equal to ED25519_KEY_SIZE nor ED25519_PRV_KEY_SIZE, or pubSz is less than ED25519_PUB_KEY_SIZE.

Example

```
int ret;
byte priv[] = { initialize with 32 byte private key };
byte pub[]  = { initialize with the corresponding public key };

ed25519_key key;
wc_ed25519_init_key(&key);
ret = wc_ed25519_import_private_key(priv, sizeof(priv), pub, sizeof(pub),
        &key);
if (ret != 0) {
    // error importing key
}
```

```
int wc_ed25519_import_private_key_ex(
    const byte * priv,
    word32 privSz,
    const byte * pub,
    word32 pubSz,
    ed25519_key * key,
    int trusted
```

)

This function imports a public/private Ed25519 key pair from a pair of buffers. This function will handle both compressed and uncompressed keys. The public is checked against private key if not trusted.

Parameters:

- **priv** Pointer to the buffer containing the private key.
- **privSz** Length of the private key.
- **pub** Pointer to the buffer containing the public key.
- **pubSz** Length of the public key.
- **key** Pointer to the ed25519_key object in which to store the imported private/public key pair.
- **trusted** Public key data is trusted or not.

See:

- wc_ed25519_import_public
- wc_ed25519_import_public_ex
- wc_ed25519_import_private_only
- wc_ed25519_import_private_key
- wc_ed25519_export_private

Return:

- 0 Returned on successfully importing the ed25519_key.
- BAD_FUNC_ARG Returned if priv or key evaluate to NULL; or if either privSz is not equal to ED25519_KEY_SIZE nor ED25519_PRV_KEY_SIZE, or pubSz is less than ED25519_PUB_KEY_SIZE.

Example

```
int ret;
byte priv[] = { initialize with 32 byte private key };
byte pub[]  = { initialize with the corresponding public key };

ed25519_key key;
wc_ed25519_init_key(&key);
ret = wc_ed25519_import_private_key_ex(priv, sizeof(priv), pub, sizeof(pub),
        &key, 1);
if (ret != 0) {
    // error importing key
}
```

```
int wc_ed25519_export_public(
    const ed25519_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports the public key from an ed25519_key structure. It stores the public key in the buffer out, and sets the bytes written to this buffer in outLen.

Parameters:

- **key** Pointer to an ed25519_key structure from which to export the public key.
- **out** Pointer to the buffer in which to store the public key.
- **outLen** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the public key.

See:

- wc_ed25519_import_public
- wc_ed25519_import_public_ex
- wc_ed25519_export_private
- wc_ed25519_export_private_only

Return:

- 0 Returned upon successfully exporting the public key.
- BAD_FUNC_ARG Returned if any of the input values evaluate to NULL.
- BUFFER_E Returned if the buffer provided is not large enough to store the public key. Upon returning this error, the function sets the size required in outLen.

Example

```
int ret;
ed25519_key key;
// initialize key, make key

char pub[32];
word32 pubSz = sizeof(pub);

ret = wc_ed25519_export_public(&key, pub, &pubSz);
if (ret != 0) {
    // error exporting public key
}
```

```
int wc_ed25519_export_private_only(
    const ed25519_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports only the private key from an ed25519_key structure. It stores the private key in the buffer out, and sets the bytes written to this buffer in outLen.

Parameters:

- **key** Pointer to an ed25519_key structure from which to export the private key.
- **out** Pointer to the buffer in which to store the private key.

- **outLen** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the private key.

See:

- wc_ed25519_export_public
- wc_ed25519_export_private
- wc_ed25519_import_private_key
- wc_ed25519_import_private_key_ex

Return:

- 0 Returned upon successfully exporting the private key.
- BAD_FUNC_ARG Returned if any of the input values evaluate to NULL.
- BUFFER_E Returned if the buffer provided is not large enough to store the private key.

Example

```
int ret;
ed25519_key key;
// initialize key, make key

char priv[32]; // 32 bytes because only private key
word32 privSz = sizeof(priv);
ret = wc_ed25519_export_private_only(&key, priv, &privSz);
if (ret != 0) {
    // error exporting private key
}
```

```
int wc_ed25519_export_private(
    const ed25519_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports the key pair from an ed25519_key structure. It stores the key pair in the buffer out, and sets the bytes written to this buffer in outLen.

Parameters:

- **key** Pointer to an ed25519_key structure from which to export the key pair.
- **out** Pointer to the buffer in which to store the key pair.
- **outLen** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the key pair.

See:

- wc_ed25519_import_private_key
- wc_ed25519_import_private_key_ex
- wc_ed25519_export_private_only

Return:

- 0 Returned upon successfully exporting the key pair.
- BAD_FUNC_ARG Returned if any of the input values evaluate to NULL.
- BUFFER_E Returned if the buffer provided is not large enough to store the key pair.

Example

```
ed25519_key key;
wc_ed25519_init(&key);

WC_RNG rng;
wc_InitRng(&rng);

wc_ed25519_make_key(&rng, 32, &key); // initialize 32 byte Ed25519 key

byte out[64]; // out needs to be a sufficient buffer size
word32 outLen = sizeof(out);
int key_size = wc_ed25519_export_private(&key, out, &outLen);
if (key_size == BUFFER_E) {
    // Check size of out compared to outLen to see if function reset outLen
}
```

```
int wc_ed25519_export_key(
    const ed25519_key * key,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz
)
```

This function exports the private and public key separately from an ed25519_key structure. It stores the private key in the buffer priv, and sets the bytes written to this buffer in privSz. It stores the public key in the buffer pub, and sets the bytes written to this buffer in pubSz.

Parameters:

- **key** Pointer to an ed25519_key structure from which to export the key pair.
- **priv** Pointer to the buffer in which to store the private key.
- **privSz** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the private key.
- **pub** Pointer to the buffer in which to store the public key.
- **pubSz** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the public key.

See:

- wc_ed25519_export_private
- wc_ed25519_export_public

Return:

- 0 Returned upon successfully exporting the key pair.
- BAD_FUNC_ARG Returned if any of the input values evaluate to NULL.
- BUFFER_E Returned if the buffer provided is not large enough to store the key pair.
- PUBLIC_KEY_E the given key only has a private key present.

Example

```
int ret;
ed25519_key key;
// initialize key, make key

char pub[32];
word32 pubSz = sizeof(pub);
char priv[32];
word32 privSz = sizeof(priv);

ret = wc_ed25519_export_key(&key, priv, &pubSz, pub, &pubSz);
if (ret != 0) {
    // error exporting public key
}
```

```
int wc_ed25519_check_key(
    ed25519_key * key
)
```

This function checks the public key in ed25519_key structure matches the private key.

Parameters:

- **key** Pointer to an ed25519_key structure holding a private and public key.

See:

- wc_ed25519_import_private_key
- wc_ed25519_import_private_key_ex

Return:

- 0 Returned if the private and public key matched.
- BAD_FUNC_ARG Returned if the given key is NULL.
- PUBLIC_KEY_E Returned if the no public key available or is invalid.

Example

```
int ret;
byte priv[] = { initialize with 57 byte private key };
byte pub[]  = { initialize with the corresponding public key };
```

```
ed25519_key key;
wc_ed25519_init_key(&key);
wc_ed25519_import_private_key_ex(priv, sizeof(priv), pub, sizeof(pub), &key,
    1);
ret = wc_ed25519_check_key(&key);
if (ret != 0) {
    // error checking key
}
```

```
int wc_ed25519_size(
    const ed25519_key * key
)
```

This function returns the size of an Ed25519 - 32 bytes.

Parameters:

- **key** Pointer to an ed25519_key structure for which to get the key size.

See: wc_ed25519_make_key

Return:

- ED25519_KEY_SIZE The size of a valid private key (32 bytes).
- BAD_FUNC_ARG Returned if the given key is NULL.

Example

```
int keySz;
ed25519_key key;
// initialize key, make key
keySz = wc_ed25519_size(&key);
if (keySz == 0) {
    // error determining key size
}
```

```
int wc_ed25519_priv_size(
    const ed25519_key * key
)
```

This function returns the private key size (secret + public) in bytes.

Parameters:

- **key** Pointer to an ed25519_key structure for which to get the key size.

See: wc_ed25519_pub_size

Return:

- ED25519_PRV_KEY_SIZE The size of the private key (64 bytes).
- BAD_FUNC_ARG Returned if key argument is NULL.

Example

```
ed25519_key key;
wc_ed25519_init(&key);

WC_RNG rng;
wc_InitRng(&rng);

wc_ed25519_make_key(&rng, 32, &key); // initialize 32 byte Ed25519 key
int key_size = wc_ed25519_priv_size(&key);
```

```
int wc_ed25519_pub_size(
    const ed25519_key * key
)
```

This function returns the compressed key size in bytes (public key).

Parameters:

- **key** Pointer to an ed25519_key structure for which to get the key size.

See: wc_ed25519_priv_size

Return:

- ED25519_PUB_KEY_SIZE The size of the compressed public key (32 bytes).
- BAD_FUNC_ARG Returns if key argument is NULL.

Example

```
ed25519_key key;
wc_ed25519_init(&key);
WC_RNG rng;
wc_InitRng(&rng);

wc_ed25519_make_key(&rng, 32, &key); // initialize 32 byte Ed25519 key
int key_size = wc_ed25519_pub_size(&key);
```

```
int wc_ed25519_sig_size(
    const ed25519_key * key
)
```

This function returns the size of an Ed25519 signature (64 in bytes).

Parameters:

- **key** Pointer to an ed25519_key structure for which to get the signature size.

See: wc_ed25519_sign_msg

Return:

- ED25519_SIG_SIZE The size of an Ed25519 signature (64 bytes).
- BAD_FUNC_ARG Returns if key argument is NULL.

Example

```
int sigSz;
ed25519_key key;
// initialize key, make key

sigSz = wc_ed25519_sig_size(&key);
if (sigSz == 0) {
    // error determining sig size
}
```

```
int wc_ed25519_sign_msg_ex(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen,
    ed25519_key * key,
    byte type,
    const byte * context,
    byte contextLen
)
```

Signs message with extended parameters.

Parameters:

- **in** Input message
- **inLen** Input message length
- **out** Output signature buffer
- **outLen** Output signature length pointer
- **key** Ed25519 key
- **type** Signature type
- **context** Context buffer
- **contextLen** Context length

See: wc_ed25519_sign_msg

Return:

- 0 on success
- negative on failure

Example

```
byte msg[] = "message";
byte sig[ED25519_SIG_SIZE];
word32 sigLen = sizeof(sig);
int ret = wc_ed25519_sign_msg_ex(msg, sizeof(msg), sig, &sigLen,
                                 &key, Ed25519, NULL, 0);
```

```
int wc_ed25519_verify_msg_ex(
    const byte * sig,
    word32 sigLen,
    const byte * msg,
    word32 msgLen,
    int * res,
    ed25519_key * key,
    byte type,
    const byte * context,
    byte contextLen
)
```

Verifies signature with extended parameters.

Parameters:

- **sig** Signature buffer
- **sigLen** Signature length
- **msg** Message buffer
- **msgLen** Message length
- **res** Verification result pointer
- **key** Ed25519 key
- **type** Signature type
- **context** Context buffer
- **contextLen** Context length

See: wc_ed25519_verify_msg

Return:

- 0 on success
- negative on failure

Example

```
byte msg[] = "message";
byte sig[ED25519_SIG_SIZE];
int res;
int ret = wc_ed25519_verify_msg_ex(sig, sizeof(sig), msg,
                                   sizeof(msg), &res, &key,
                                   Ed25519, NULL, 0);
```

```
int wc_ed25519_verify_msg_init(
    const byte * sig,
    word32 sigLen,
    ed25519_key * key,
    byte type,
    const byte * context,
    byte contextLen
)
```

Initializes streaming verification.

Parameters:

- **sig** Signature buffer
- **sigLen** Signature length
- **key** Ed25519 key
- **type** Signature type
- **context** Context buffer
- **contextLen** Context length

See:

- wc_ed25519_verify_msg_update
- wc_ed25519_verify_msg_final

Return:

- 0 on success
- negative on failure

Example

```
byte sig[ED25519_SIG_SIZE];
int ret = wc_ed25519_verify_msg_init(sig, sizeof(sig), &key,
                                     Ed25519, NULL, 0);
```

```
int wc_ed25519_verify_msg_update(
    const byte * msgSegment,
    word32 msgSegmentLen,
    ed25519_key * key
)
```

Updates streaming verification with message segment.

Parameters:

- **msgSegment** Message segment buffer
- **msgSegmentLen** Message segment length
- **key** Ed25519 key

See:

- wc_ed25519_verify_msg_init
- wc_ed25519_verify_msg_final

Return:

- 0 on success
- negative on failure

Example

```
byte msgPart[] = "part";
int ret = wc_ed25519_verify_msg_update(msgPart, sizeof(msgPart),
                                       &key);
```

```
int wc_ed25519_verify_msg_final(
    const byte * sig,
    word32 sigLen,
    int * res,
    ed25519_key * key
)
```

Finalizes streaming verification.

Parameters:

- **sig** Signature buffer
- **sigLen** Signature length
- **res** Verification result pointer
- **key** Ed25519 key

See:

- wc_ed25519_verify_msg_init
- wc_ed25519_verify_msg_update

Return:

- 0 on success
- negative on failure

Example

```
byte sig[ED25519_SIG_SIZE];
int res;
int ret = wc_ed25519_verify_msg_final(sig, sizeof(sig), &res,
                                      &key);
```

```
int wc_ed25519_init_ex(
    ed25519_key * key,
    void * heap,
    int devId
)
```

Initializes Ed25519 key with extended parameters.

Parameters:

- **key** Ed25519 key structure
- **heap** Heap hint for memory allocation
- **devId** Device ID for hardware acceleration

See: wc_ed25519_init

Return:

- 0 on success
- negative on failure

Example

```
ed25519_key key;
int ret = wc_ed25519_init_ex(&key, NULL, INVALID_DEVID);
```

```
int wc_ed25519_init_id(
    ed25519_key * key,
    const unsigned char * id,
    int len,
    void * heap,
    int devId
)
```

Initializes an ed25519_key object that references a private key held by a crypto callback device, identified by id. Only available when built with WOLF_PRIVATE_KEY_ID.

Parameters:

- **key** Pointer to the ed25519_key to initialize.
- **id** Key identifier bytes.

- **len** Length of id in bytes.
- **heap** Heap hint. May be NULL.
- **devId** Device identifier for the crypto callback.

See:

- wc_ed25519_init_ex
- wc_ed25519_init_label

Return:

- 0 on success.
- BAD_FUNC_ARG if key is NULL, or id is NULL and len is greater than 0.
- BUFFER_E if len is negative or greater than ED25519_MAX_ID_LEN.

Example

```
ed25519_key key;
unsigned char id[] = { 0x01, 0x02, 0x03, 0x04 };
int ret = wc_ed25519_init_id(&key, id, sizeof(id), NULL, devId);
```

```
int wc_ed25519_init_label(
    ed25519_key * key,
    const char * label,
    void * heap,
    int devId
)
```

Initializes an ed25519_key object that references a private key held by a crypto callback device, identified by label. Only available when built with WOLF_PRIVATE_KEY_ID.

Parameters:

- **key** Pointer to the ed25519_key to initialize.
- **label** NUL-terminated label string.
- **heap** Heap hint. May be NULL.
- **devId** Device identifier for the crypto callback.

See:

- wc_ed25519_init_ex
- wc_ed25519_init_id

Return:

- 0 on success.
- BAD_FUNC_ARG if key or label is NULL.
- BUFFER_E if label is empty or longer than ED25519_MAX_LABEL_LEN.

Example

```
ed25519_key key;
int ret = wc_ed25519_init_label(&key, "my-key", NULL, devId);
```

```
ed25519_key * wc_ed25519_new(
    void * heap,
    int devId,
    int * result_code
)
```

Allocates and initializes new Ed25519 key. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.

Parameters:

- **heap** Heap hint for memory allocation
- **devId** Device ID for hardware acceleration
- **result_code** Result code pointer

See: wc_ed25519_delete

Return:

- ed25519_key pointer on success
- NULL on failure

Note: This API is only available when WC_NO_CONSTRUCTORS is not defined. WC_NO_CONSTRUCTORS is automatically defined when WOLFSSL_NO_MALLOC is defined.

Example

```
int result;
ed25519_key* key = wc_ed25519_new(NULL, INVALID_DEVID, &result);
```

```
int wc_ed25519_delete(
    ed25519_key * key,
    ed25519_key ** key_p
)
```

Frees and deletes Ed25519 key. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.

Parameters:

- **key** Ed25519 key to delete
- **key_p** Pointer to key pointer (set to NULL after delete)

See: wc_ed25519_new

Return:

- 0 on success
- negative on failure

Note: This API is only available when WC_NO_CONSTRUCTORS is not defined. WC_NO_CONSTRUCTORS is automatically defined when WOLFSSL_NO_MALLOC is defined.

Example

```
ed25519_key* key = wc_ed25519_new(NULL, INVALID_DEVID, NULL);
int ret = wc_ed25519_delete(key, &key);
```

B.26 Algorithms - ED448

B.25.2.36 function wc_ed25519_delete

B.26.1 Functions

	Name
int	**wc_ed448_make_public**(ed448_key * key, unsigned char * pubKey, word32 pubKeySz)This function generates the Ed448 public key from the private key. It stores the public key in the buffer pubKey, and sets the bytes written to this buffer in pubKeySz. If the key object does not already carry a public key, the derived key is also stored in it, so that the key can be used for signing.
int	**wc_ed448_make_key**(WC_RNG * rng, int keysize, ed448_key * key)This function generates a new Ed448 key and stores it in key.
int	**wc_ed448_sign_msg**(const byte * in, word32 inLen, byte * out, word32 * outLen, ed448_key * key, const byte * context, byte contextLen)This function signs a message using an ed448_key object to guarantee authenticity.
int	**wc_ed448ph_sign_hash**(const byte * hash, word32 hashLen, byte * out, word32 * outLen, ed448_key * key, const byte * context, byte contextLen)This function signs a message digest using an ed448_key object to guarantee authenticity. The context is included as part of the data signed. The hash is the pre-hashed message before signature calculation.

	Name
int	**wc_ed448ph_sign_msg**(const byte * in, word32 inLen, byte * out, word32 * outLen, ed448_key * key, const byte * context, byte contextLen)This function signs a message using an ed448_key object to guarantee authenticity. The context is included as part of the data signed. The message is pre-hashed before signature calculation.
int	**wc_ed448_verify_msg**(const byte * sig, word32 siglen, const byte * msg, word32 msgLen, int * res, ed448_key * key, const byte * context, byte contextLen)This function verifies the Ed448 signature of a message to ensure authenticity. The context is included as part of the data verified. The answer is returned through res, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.
int	**wc_ed448ph_verify_hash**(const byte * sig, word32 siglen, const byte * hash, word32 hashlen, int * res, ed448_key * key, const byte * context, byte contextLen)This function verifies the Ed448 signature of the digest of a message to ensure authenticity. The context is included as part of the data verified. The hash is the pre-hashed message before signature calculation. The answer is returned through res, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.
int	**wc_ed448ph_verify_msg**(const byte * sig, word32 siglen, const byte * msg, word32 msgLen, int * res, ed448_key * key, const byte * context, byte contextLen)This function verifies the Ed448 signature of a message to ensure authenticity. The context is included as part of the data verified. The message is pre-hashed before verification. The answer is returned through res, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.
int	**wc_ed448_init**(ed448_key * key)This function initializes an ed448_key object for future use with message verification.
int	**wc_ed448_init_id**(ed448_key * key, const unsigned char * id, int len, void * heap, int devId)Initializes an ed448_key object that references a private key held by a crypto callback device, identified by id. Only available when built with WOLF_PRIVATE_KEY_ID.

	Name
int	**wc_ed448_init_label**(ed448_key * key, const char * label, void * heap, int devId)Initializes an ed448_key object that references a private key held by a crypto callback device, identified by label. Only available when built with WOLF_PRIVATE_KEY_ID.
void	**wc_ed448_free**(ed448_key * key)This function frees an Ed448 object after it has been used.
int	**wc_ed448_import_public**(const byte * in, word32 inLen, ed448_key * key)This function imports a public ed448_key pair from a buffer containing the public key. This function will handle both compressed and uncompressed keys. The public key is checked that it matches the private key when one is present.
int	**wc_ed448_import_public_ex**(const byte * in, word32 inLen, ed448_key * key, int trusted)This function imports a public ed448_key pair from a buffer containing the public key. This function will handle both compressed and uncompressed keys. Check public key matches private key, when present, when not trusted.
int	**wc_ed448_import_private_only**(const byte * priv, word32 privSz, ed448_key * key)This function imports an Ed448 private key only from a buffer.
int	**wc_ed448_import_private_key**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ed448_key * key)This function imports a public/private Ed448 key pair from a pair of buffers. This function will handle both compressed and uncompressed keys.
int	**wc_ed448_import_private_key_ex**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ed448_key * key, int trusted)This function imports a public/private Ed448 key pair from a pair of buffers. This function will handle both compressed and uncompressed keys. The public is checked against private key if not trusted.
int	**wc_ed448_export_public**(const ed448_key * key, byte * out, word32 * outLen)This function exports the private key from an ed448_key structure. It stores the public key in the buffer out, and sets the bytes written to this buffer in outLen.
int	**wc_ed448_export_private_only**(const ed448_key * key, byte * out, word32 * outLen)This function exports only the private key from an ed448_key structure. It stores the private key in the buffer out, and sets the bytes written to this buffer in outLen.

	Name
int	**wc_ed448_export_private**(const ed448_key * key, byte * out, word32 * outLen)This function exports the key pair from an ed448_key structure. It stores the key pair in the buffer out, and sets the bytes written to this buffer in outLen.
int	**wc_ed448_export_key**(const ed448_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz)This function exports the private and public key separately from an ed448_key structure. It stores the private key in the buffer priv, and sets the bytes written to this buffer in privSz. It stores the public key in the buffer pub, and sets the bytes written to this buffer in pubSz.
int	**wc_ed448_check_key**(ed448_key * key)This function checks the public key in ed448_key structure matches the private key.
int	**wc_ed448_size**(const ed448_key * key)This function returns the size of an Ed448 private key - 57 bytes.
int	**wc_ed448_priv_size**(const ed448_key * key)This function returns the private key size (secret + public) in bytes.
int	**wc_ed448_pub_size**(const ed448_key * key)This function returns the compressed key size in bytes (public key).
int	**wc_ed448_sig_size**(const ed448_key * key)This function returns the size of an Ed448 signature (114 in bytes).

B.26.2 Functions Documentation

```
int wc_ed448_make_public(
    ed448_key * key,
    unsigned char * pubKey,
    word32 pubKeySz
)
```

This function generates the Ed448 public key from the private key. It stores the public key in the buffer pubKey, and sets the bytes written to this buffer in pubKeySz. If the key object does not already carry a public key, the derived key is also stored in it, so that the key can be used for signing.

Parameters:

- **key** Pointer to the ed448_key for which to generate a key.
- **pubKey** Pointer to the buffer in which to store the public key.
- **pubKeySz** Size of the pubKey buffer in bytes.

See:

- wc_ed448_init
- wc_ed448_import_private_only
- wc_ed448_make_key

Return:

- 0 Returned upon successfully making the public key.
- BAD_FUNC_ARG Returned ifi key or pubKey evaluate to NULL, or if the specified key size is not 57 bytes (Ed448 has 57 byte keys).
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
int ret;

ed448_key key;
byte priv[] = { initialize with 57 byte private key };
byte pub[57];
word32 pubSz = sizeof(pub);

wc_ed448_init(&key);
wc_ed448_import_private_only(priv, sizeof(priv), &key);
ret = wc_ed448_make_public(&key, pub, pubSz);
if (ret != 0) {
    // error making public key
}
```

```
int wc_ed448_make_key(
    WC_RNG * rng,
    int keysize,
    ed448_key * key
)
```

This function generates a new Ed448 key and stores it in key.

Parameters:

- **rng** Pointer to an initialized RNG object with which to generate the key.
- **keysize** Length of key to generate. Should always be 57 for Ed448.
- **key** Pointer to the ed448_key for which to generate a key.

See: wc_ed448_init

Return:

- 0 Returned upon successfully making an ed448_key.
- BAD_FUNC_ARG Returned if rng or key evaluate to NULL, or if the specified key size is not 57 bytes (Ed448 has 57 byte keys).
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
int ret;

WC_RNG rng;
ed448_key key;

wc_InitRng(&rng);
wc_ed448_init(&key);
ret = wc_ed448_make_key(&rng, 57, &key);
if (ret != 0) {
    // error making key
}
```

```
int wc_ed448_sign_msg(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen,
    ed448_key * key,
    const byte * context,
    byte contextLen
)
```

This function signs a message using an ed448_key object to guarantee authenticity.

Parameters:

- **in** Pointer to the buffer containing the message to sign.
- **inLen** Length of the message to sign.
- **out** Buffer in which to store the generated signature.
- **outLen** Maximum length of the output buffer. Will store the bytes written to out upon successfully generating a message signature.
- **key** Pointer to a private ed448_key with which to generate the signature.
- **context** Pointer to the buffer containing the context for which message is being signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed448ph_sign_hash
- wc_ed448ph_sign_msg
- wc_ed448_verify_msg

Return:

- 0 Returned upon successfully generating a signature for the message.
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the output buffer is too small to store the generated signature.
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
ed448_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[114]; // will hold generated signature
sigSz = sizeof(sig);
byte message[] = { initialize with message };

wc_InitRng(&rng); // initialize rng
wc_ed448_init(&key); // initialize key
wc_ed448_make_key(&rng, 57, &key); // make public/private key pair
ret = wc_ed448_sign_msg(message, sizeof(message), sig, &sigSz, &key);
if (ret != 0 ) {
    // error generating message signature
}
```

```
int wc_ed448ph_sign_hash(
    const byte * hash,
    word32 hashLen,
    byte * out,
    word32 * outLen,
    ed448_key * key,
    const byte * context,
    byte contextLen
)
```

This function signs a message digest using an ed448_key object to guarantee authenticity. The context is included as part of the data signed. The hash is the pre-hashed message before signature calculation.

Parameters:

- **hash** Pointer to the buffer containing the hash of the message to sign.
- **hashLen** Length of the hash of the message to sign.
- **out** Buffer in which to store the generated signature.
- **outLen** Maximum length of the output buffer. Will store the bytes written to out upon successfully generating a message signature.
- **key** Pointer to a private ed448_key with which to generate the signature.
- **context** Pointer to the buffer containing the context for which message is being signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed448_sign_msg
- wc_ed448ph_sign_msg
- wc_ed448ph_verify_hash

Return:

- 0 Returned upon successfully generating a signature for the message digest.
- BAD_FUNC_ARG Returned any of the input parameters evaluate to NULL, or if the output buffer is too small to store the generated signature.
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
ed448_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[114]; // will hold generated signature
sigSz = sizeof(sig);
byte hash[] = { initialize hash of message };
byte context[] = { initialize with context of signing };

wc_InitRng(&rng); // initialize rng
wc_ed448_init(&key); // initialize key
wc_ed448_make_key(&rng, 57, &key); // make public/private key pair
ret = wc_ed448ph_sign_hash(hash, sizeof(hash), sig, &sigSz, &key,
        context, sizeof(context));
if (ret != 0) {
    // error generating message signature
}
```

```
int wc_ed448ph_sign_msg(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen,
    ed448_key * key,
    const byte * context,
    byte contextLen
)
```

This function signs a message using an ed448_key object to guarantee authenticity. The context is included as part of the data signed. The message is pre-hashed before signature calculation.

Parameters:

- **in** Pointer to the buffer containing the message to sign.
- **inLen** Length of the message to sign.
- **out** Buffer in which to store the generated signature.
- **outLen** Maximum length of the output buffer. Will store the bytes written to out upon successfully generating a message signature.
- **key** Pointer to a private ed448_key with which to generate the signature.
- **context** Pointer to the buffer containing the context for which message is being signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed448_sign_msg
- wc_ed448ph_sign_hash
- wc_ed448ph_verify_msg

Return:

- 0 Returned upon successfully generating a signature for the message.
- BAD_FUNC_ARG Returned any of the input parameters evaluate to NULL, or if the output buffer is too small to store the generated signature.
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
ed448_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[114]; // will hold generated signature
sigSz = sizeof(sig);
byte message[] = { initialize with message };
byte context[] = { initialize with context of signing };

wc_InitRng(&rng); // initialize rng
wc_ed448_init(&key); // initialize key
wc_ed448_make_key(&rng, 57, &key); // make public/private key pair
ret = wc_ed448ph_sign_msg(message, sizeof(message), sig, &sigSz, &key,
        context, sizeof(context));
if (ret != 0) {
    // error generating message signature
}
```

```
int wc_ed448_verify_msg(
    const byte * sig,
    word32 siglen,
    const byte * msg,
    word32 msgLen,
    int * res,
    ed448_key * key,
    const byte * context,
    byte contextLen
)
```

This function verifies the Ed448 signature of a message to ensure authenticity. The context is included as part of the data verified. The answer is returned through res, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

Parameters:

- **sig** Pointer to the buffer containing the signature to verify.

- **siglen** Length of the signature to verify.
- **msg** Pointer to the buffer containing the message to verify.
- **msgLen** Length of the message to verify.
- **res** Pointer to an int that will be set to 1 for a valid signature or 0 for an invalid signature after verification completes.
- **key** Pointer to a public Ed448 key with which to verify the signature.
- **context** Pointer to the buffer containing the context for which the message was signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed448ph_verify_hash
- wc_ed448ph_verify_msg
- wc_ed448_sign_msg

Return:

- 0 Returned upon successfully performing the signature verification and authentication.
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the siglen does not match the actual length of a signature.
- SIG_VERIFY_E Returned if verification completes, but the signature generated does not match the signature provided.

Example

```
ed448_key key;
int ret, verified = 0;

byte sig[] { initialize with received signature };
byte msg[] = { initialize with message };
byte context[] = { initialize with context of signature };
// initialize key with received public key
ret = wc_ed448_verify_msg(sig, sizeof(sig), msg, sizeof(msg), &verified,
        &key, context, sizeof(context));
if (ret < 0) {
    // error performing verification
} else if (verified == 0)
    // the signature is invalid
}
```

```
int wc_ed448ph_verify_hash(
    const byte * sig,
    word32 siglen,
    const byte * hash,
    word32 hashlen,
    int * res,
    ed448_key * key,
    const byte * context,
    byte contextLen
)
```

This function verifies the Ed448 signature of the digest of a message to ensure authenticity. The context is included as part of the data verified. The hash is the pre-hashed message before signature calculation. The answer is returned through res, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

Parameters:

- **sig** Pointer to the buffer containing the signature to verify.
- **siglen** Length of the signature to verify.
- **hash** Pointer to the buffer containing the hash of the message to verify.
- **hashlen** Length of the hash to verify.
- **res** Pointer to an int that will be set to 1 for a valid signature or 0 for an invalid signature after verification completes.
- **key** Pointer to a public Ed448 key with which to verify the signature.
- **context** Pointer to the buffer containing the context for which the message was signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed448_verify_msg
- wc_ed448ph_verify_msg
- wc_ed448ph_sign_hash

Return:

- 0 Returned upon successfully performing the signature verification and authentication.
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the siglen does not match the actual length of a signature.
- SIG_VERIFY_E Returned if verification completes, but the signature generated does not match the signature provided.

Example

```
ed448_key key;
int ret, verified = 0;

byte sig[] { initialize with received signature };
byte hash[] = { initialize hash of message };
byte context[] = { initialize with context of signature };
// initialize key with received public key
ret = wc_ed448ph_verify_hash(sig, sizeof(sig), hash, sizeof(hash),
        &verified, &key, context, sizeof(context));
if (ret < 0) {
    // error performing verification
} else if (verified == 0)
    // the signature is invalid
}
```

```
int wc_ed448ph_verify_msg(
    const byte * sig,
```

```
    word32 siglen,
    const byte * msg,
    word32 msgLen,
    int * res,
    ed448_key * key,
    const byte * context,
    byte contextLen
)
```

This function verifies the Ed448 signature of a message to ensure authenticity. The context is included as part of the data verified. The message is pre-hashed before verification. The answer is returned through res, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

Parameters:

- **sig** Pointer to the buffer containing the signature to verify.
- **siglen** Length of the signature to verify.
- **msg** Pointer to the buffer containing the message to verify.
- **msgLen** Length of the message to verify.
- **res** Pointer to an int that will be set to 1 for a valid signature or 0 for an invalid signature after verification completes.
- **key** Pointer to a public Ed448 key with which to verify the signature.
- **context** Pointer to the buffer containing the context for which the message was signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed448_verify_msg
- wc_ed448ph_verify_hash
- wc_ed448ph_sign_msg

Return:

- 0 Returned upon successfully performing the signature verification and authentication.
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the siglen does not match the actual length of a signature.
- SIG_VERIFY_E Returned if verification completes, but the signature generated does not match the signature provided.

Example

```
ed448_key key;
int ret, verified = 0;

byte sig[] { initialize with received signature };
byte msg[] = { initialize with message };
byte context[] = { initialize with context of signature };
// initialize key with received public key
ret = wc_ed448ph_verify_msg(sig, sizeof(sig), msg, sizeof(msg), &verified,
        &key, context, sizeof(context));
if (ret < 0) {
    // error performing verification
```

```
} else if (verified == 0)
    // the signature is invalid
}
```

```
int wc_ed448_init(
    ed448_key * key
)
```

This function initializes an ed448_key object for future use with message verification.

Parameters:

- **key** Pointer to the ed448_key object to initialize.

See:

- wc_ed448_make_key
- wc_ed448_free

Return:

- 0 Returned upon successfully initializing the ed448_key object.
- BAD_FUNC_ARG Returned if key is NULL.

Example

```
ed448_key key;
wc_ed448_init(&key);
```

```
int wc_ed448_init_id(
    ed448_key * key,
    const unsigned char * id,
    int len,
    void * heap,
    int devId
)
```

Initializes an ed448_key object that references a private key held by a crypto callback device, identified by id. Only available when built with WOLF_PRIVATE_KEY_ID.

Parameters:

- **key** Pointer to the ed448_key to initialize.
- **id** Key identifier bytes.
- **len** Length of id in bytes.
- **heap** Heap hint. May be NULL.
- **devId** Device identifier for the crypto callback.

See:

- wc_ed448_init_ex
- wc_ed448_init_label

Return:

- 0 on success.
- BAD_FUNC_ARG if key is NULL, or id is NULL and len is greater than 0.
- BUFFER_E if len is negative or greater than ED448_MAX_ID_LEN.

Example

```
ed448_key key;
unsigned char id[] = { 0x01, 0x02, 0x03, 0x04 };
int ret = wc_ed448_init_id(&key, id, sizeof(id), NULL, devId);
```

```
int wc_ed448_init_label(
    ed448_key * key,
    const char * label,
    void * heap,
    int devId
)
```

Initializes an ed448_key object that references a private key held by a crypto callback device, identified by label. Only available when built with WOLF_PRIVATE_KEY_ID.

Parameters:

- **key** Pointer to the ed448_key to initialize.
- **label** NUL-terminated label string.
- **heap** Heap hint. May be NULL.
- **devId** Device identifier for the crypto callback.

See:

- wc_ed448_init_ex
- wc_ed448_init_id

Return:

- 0 on success.
- BAD_FUNC_ARG if key or label is NULL.
- BUFFER_E if label is empty or longer than ED448_MAX_LABEL_LEN.

Example

```
ed448_key key;
int ret = wc_ed448_init_label(&key, "my-key", NULL, devId);
```

```
void wc_ed448_free(
    ed448_key * key
)
```

This function frees an Ed448 object after it has been used.

Parameters:

- **key** Pointer to the ed448_key object to free

See: wc_ed448_init

Example

```
ed448_key key;
// initialize key and perform secure exchanges
...
wc_ed448_free(&key);
```

```
int wc_ed448_import_public(
    const byte * in,
    word32 inLen,
    ed448_key * key
)
```

This function imports a public ed448_key pair from a buffer containing the public key. This function will handle both compressed and uncompressed keys. The public key is checked that it matches the private key when one is present.

Parameters:

- **in** Pointer to the buffer containing the public key.
- **inLen** Length of the buffer containing the public key.
- **key** Pointer to the ed448_key object in which to store the public key.

See:

- wc_ed448_import_public_ex
- wc_ed448_import_private_key
- wc_ed448_import_private_key_ex
- wc_ed448_export_public

Return:

- 0 Returned on successfully importing the ed448_key.

- BAD_FUNC_ARG Returned if in or key evaluate to NULL, or inLen is less than the size of an Ed448 key.

Example

```
int ret;
byte pub[] = { initialize Ed448 public key };

ed_448 key;
wc_ed448_init_key(&key);
ret = wc_ed448_import_public(pub, sizeof(pub), &key);
if (ret != 0) {
    // error importing key
}
```

```
int wc_ed448_import_public_ex(
    const byte * in,
    word32 inLen,
    ed448_key * key,
    int trusted
)
```

This function imports a public ed448_key pair from a buffer containing the public key. This function will handle both compressed and uncompressed keys. Check public key matches private key, when present, when not trusted.

Parameters:

- **in** Pointer to the buffer containing the public key.
- **inLen** Length of the buffer containing the public key.
- **key** Pointer to the ed448_key object in which to store the public key.
- **trusted** Public key data is trusted or not.

See:

- wc_ed448_import_public
- wc_ed448_import_private_key
- wc_ed448_import_private_key_ex
- wc_ed448_export_public

Return:

- 0 Returned on successfully importing the ed448_key.
- BAD_FUNC_ARG Returned if in or key evaluate to NULL, or inLen is less than the size of an Ed448 key.

Example

```
int ret;
byte pub[] = { initialize Ed448 public key };

ed_448 key;
wc_ed448_init_key(&key);
ret = wc_ed448_import_public_ex(pub, sizeof(pub), &key, 1);
if (ret != 0) {
    // error importing key
}
```

```
int wc_ed448_import_private_only(
    const byte * priv,
    word32 privSz,
    ed448_key * key
)
```

This function imports an Ed448 private key only from a buffer.

Parameters:

- **priv** Pointer to the buffer containing the private key.
- **privSz** Length of the private key.
- **key** Pointer to the ed448_key object in which to store the imported private key.

See:

- wc_ed448_import_public
- wc_ed448_import_public_ex
- wc_ed448_import_private_key
- wc_ed448_import_private_key_ex
- wc_ed448_export_private_only

Return:

- 0 Returned on successfully importing the Ed448 private key.
- BAD_FUNC_ARG Returned if in or key evaluate to NULL, or if privSz is less than ED448_KEY_SIZE.

Example

```
int ret;
byte priv[] = { initialize with 57 byte private key };

ed448_key key;
wc_ed448_init_key(&key);
ret = wc_ed448_import_private_only(priv, sizeof(priv), &key);
if (ret != 0) {
    // error importing private key
}
```

```
int wc_ed448_import_private_key(
    const byte * priv,
    word32 privSz,
    const byte * pub,
    word32 pubSz,
    ed448_key * key
)
```

This function imports a public/private Ed448 key pair from a pair of buffers. This function will handle both compressed and uncompressed keys.

Parameters:

- **priv** Pointer to the buffer containing the private key.
- **privSz** Length of the private key.
- **pub** Pointer to the buffer containing the public key.
- **pubSz** Length of the public key.
- **key** Pointer to the ed448_key object in which to store the imported private/public key pair.

See:

- wc_ed448_import_public
- wc_ed448_import_public_ex
- wc_ed448_import_private_only
- wc_ed448_import_private_key_ex
- wc_ed448_export_private

Return:

- 0 Returned on successfully importing the Ed448 key.
- BAD_FUNC_ARG Returned if in or key evaluate to NULL, or if either privSz is less than ED448_KEY_SIZE or pubSz is less than ED448_PUB_KEY_SIZE.

Example

```
int ret;
byte priv[] = { initialize with 57 byte private key };
byte pub[]  = { initialize with the corresponding public key };

ed448_key key;
wc_ed448_init_key(&key);
ret = wc_ed448_import_private_key(priv, sizeof(priv), pub, sizeof(pub),
        &key);
if (ret != 0) {
    // error importing key
}
```

```
int wc_ed448_import_private_key_ex(
    const byte * priv,
    word32 privSz,
```

```
    const byte * pub,
    word32 pubSz,
    ed448_key * key,
    int trusted
)
```

This function imports a public/private Ed448 key pair from a pair of buffers. This function will handle both compressed and uncompressed keys. The public is checked against private key if not trusted.

Parameters:

- **priv** Pointer to the buffer containing the private key.
- **privSz** Length of the private key.
- **pub** Pointer to the buffer containing the public key.
- **pubSz** Length of the public key.
- **key** Pointer to the ed448_key object in which to store the imported private/public key pair.
- **trusted** Public key data is trusted or not.

See:

- wc_ed448_import_public
- wc_ed448_import_public_ex
- wc_ed448_import_private_only
- wc_ed448_import_private_key
- wc_ed448_export_private

Return:

- 0 Returned on successfully importing the Ed448 key.
- BAD_FUNC_ARG Returned if in or key evaluate to NULL, or if either privSz is less than ED448_KEY_SIZE or pubSz is less than ED448_PUB_KEY_SIZE.

Example

```
int ret;
byte priv[] = { initialize with 57 byte private key };
byte pub[]  = { initialize with the corresponding public key };

ed448_key key;
wc_ed448_init_key(&key);
ret = wc_ed448_import_private_key_ex(priv, sizeof(priv), pub, sizeof(pub),
        &key, 1);
if (ret != 0) {
    // error importing key
}
```

```
int wc_ed448_export_public(
    const ed448_key * key,
    byte * out,
    word32 * outLen
```

```
)
```

This function exports the private key from an ed448_key structure. It stores the public key in the buffer out, and sets the bytes written to this buffer in outLen.

Parameters:

- **key** Pointer to an ed448_key structure from which to export the public key.
- **out** Pointer to the buffer in which to store the public key.
- **outLen** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the public key.

See:

- wc_ed448_import_public
- wc_ed448_import_public_ex
- wc_ed448_export_private_only

Return:

- 0 Returned upon successfully exporting the public key.
- BAD_FUNC_ARG Returned if any of the input values evaluate to NULL.
- BUFFER_E Returned if the buffer provided is not large enough to store the private key. Upon returning this error, the function sets the size required in outLen.
- PUBLIC_KEY_E the given key only has a private key present.

Example

```
int ret;
ed448_key key;
// initialize key, make key

char pub[57];
word32 pubSz = sizeof(pub);

ret = wc_ed448_export_public(&key, pub, &pubSz);
if (ret != 0) {
    // error exporting public key
}
```

```
int wc_ed448_export_private_only(
    const ed448_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports only the private key from an ed448_key structure. It stores the private key in the buffer out, and sets the bytes written to this buffer in outLen.

Parameters:

- **key** Pointer to an ed448_key structure from which to export the private key.
- **out** Pointer to the buffer in which to store the private key.
- **outLen** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the private key.

See:

- wc_ed448_export_public
- wc_ed448_import_private_key
- wc_ed448_import_private_key_ex

Return:

- 0 Returned upon successfully exporting the private key.
- ECC_BAD_ARG_E Returned if any of the input values evaluate to NULL.
- BUFFER_E Returned if the buffer provided is not large enough to store the private key.

Example

```
int ret;
ed448_key key;
// initialize key, make key

char priv[57]; // 57 bytes because only private key
word32 privSz = sizeof(priv);
ret = wc_ed448_export_private_only(&key, priv, &privSz);
if (ret != 0) {
    // error exporting private key
}
```

```
int wc_ed448_export_private(
    const ed448_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports the key pair from an ed448_key structure. It stores the key pair in the buffer out, and sets the bytes written to this buffer in outLen.

Parameters:

- **key** Pointer to an ed448_key structure from which to export the key pair.
- **out** Pointer to the buffer in which to store the key pair.
- **outLen** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the key pair.

See:

- wc_ed448_import_private
- wc_ed448_export_private_only

Return:

- 0 Returned upon successfully exporting the key pair.
- ECC_BAD_ARG_E Returned if any of the input values evaluate to NULL.
- BUFFER_E Returned if the buffer provided is not large enough to store the key pair.

Example

```
ed448_key key;
wc_ed448_init(&key);

WC_RNG rng;
wc_InitRng(&rng);

wc_ed448_make_key(&rng, 57, &key); // initialize 57 byte Ed448 key

byte out[114]; // out needs to be a sufficient buffer size
word32 outLen = sizeof(out);
int key_size = wc_ed448_export_private(&key, out, &outLen);
if (key_size == BUFFER_E) {
    // Check size of out compared to outLen to see if function reset outLen
}
```

```
int wc_ed448_export_key(
    const ed448_key * key,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz
)
```

This function exports the private and public key separately from an ed448_key structure. It stores the private key in the buffer priv, and sets the bytes written to this buffer in privSz. It stores the public key in the buffer pub, and sets the bytes written to this buffer in pubSz.

Parameters:

- **key** Pointer to an ed448_key structure from which to export the key pair.
- **priv** Pointer to the buffer in which to store the private key.
- **privSz** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the private key.
- **pub** Pointer to the buffer in which to store the public key.
- **pubSz** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the public key.

See:

- wc_ed448_export_private
- wc_ed448_export_public

Return:

- 0 Returned upon successfully exporting the key pair.
- ECC_BAD_ARG_E Returned if any of the input values evaluate to NULL.
- BUFFER_E Returned if the buffer provided is not large enough to store the key pair.
- PUBLIC_KEY_E the given key only has a private key present.

Example

```
int ret;
ed448_key key;
// initialize key, make key

char pub[57];
word32 pubSz = sizeof(pub);
char priv[57];
word32 privSz = sizeof(priv);

ret = wc_ed448_export_key(&key, priv, &pubSz, pub, &pubSz);
if (ret != 0) {
    // error exporting private and public key
}
```

```
int wc_ed448_check_key(
    ed448_key * key
)
```

This function checks the public key in ed448_key structure matches the private key.

Parameters:

- **key** Pointer to an ed448_key structure holding a private and public key.

See:

- wc_ed448_import_private_key
- wc_ed448_import_private_key_ex

Return:

- 0 Returned if the private and public key matched.
- BAD_FUNC_ARGS Returned if the given key is NULL.

Example

```
int ret;
byte priv[] = { initialize with 57 byte private key };
byte pub[]  = { initialize with the corresponding public key };
```

```
ed448_key key;
wc_ed448_init_key(&key);
wc_ed448_import_private_key_ex(priv, sizeof(priv), pub, sizeof(pub), &key,
    1);
ret = wc_ed448_check_key(&key);
if (ret != 0) {
    // error checking key
}
```

```
int wc_ed448_size(
    const ed448_key * key
)
```

This function returns the size of an Ed448 private key - 57 bytes.

Parameters:

- **key** Pointer to an ed448_key structure for which to get the key size.

See: wc_ed448_make_key

Return:

- ED448_KEY_SIZE The size of a valid private key (57 bytes).
- BAD_FUNC_ARGS Returned if the given key is NULL.

Example

```
int keySz;
ed448_key key;
// initialize key, make key
keySz = wc_ed448_size(&key);
if (keySz == 0) {
    // error determining key size
}
```

```
int wc_ed448_priv_size(
    const ed448_key * key
)
```

This function returns the private key size (secret + public) in bytes.

Parameters:

- **key** Pointer to an ed448_key structure for which to get the key size.

See: wc_ed448_pub_size

Return:

- ED448_PRV_KEY_SIZE The size of the private key (114 bytes).
- BAD_FUNC_ARG Returns if key argument is NULL.

Example

```
ed448_key key;
wc_ed448_init(&key);

WC_RNG rng;
wc_InitRng(&rng);

wc_ed448_make_key(&rng, 57, &key); // initialize 57 byte Ed448 key
int key_size = wc_ed448_priv_size(&key);
```

```
int wc_ed448_pub_size(
    const ed448_key * key
)
```

This function returns the compressed key size in bytes (public key).

Parameters:

- **key** Pointer to an ed448_key structure for which to get the key size.

See: wc_ed448_priv_size

Return:

- ED448_PUB_KEY_SIZE The size of the compressed public key (57 bytes).
- BAD_FUNC_ARG Returns if key argument is NULL.

Example

```
ed448_key key;
wc_ed448_init(&key);
WC_RNG rng;
wc_InitRng(&rng);

wc_ed448_make_key(&rng, 57, &key); // initialize 57 byte Ed448 key
int key_size = wc_ed448_pub_size(&key);
```

```
int wc_ed448_sig_size(
    const ed448_key * key
)
```

This function returns the size of an Ed448 signature (114 in bytes).

Parameters:

- **key** Pointer to an ed448_key structure for which to get the signature size.

See: wc_ed448_sign_msg

Return:

- ED448_SIG_SIZE The size of an Ed448 signature (114 bytes).
- BAD_FUNC_ARG Returns if key argument is NULL.

Example

```
int sigSz;
ed448_key key;
// initialize key, make key

sigSz = wc_ed448_sig_size(&key);
if (sigSz == 0) {
    // error determining sig size
}
```

B.27 Overview of ECCSI

B.26.2.26 function wc_ed448_sig_size More...

B.27.1 Detailed Description

ECCSI (Elliptic Curve-Based Certificateless Signatures for Identity-Based Encryption) is specified in RFC 6507 (https://tools.ietf.org/html/rfc6507).

In Identity-Based cryptography, there is a Key Management Service that generates keys based on an identity for a client. The private key (SSK) and public key (PVT) are delivered to the signer and the public key (PVT) only delivered to the verifier on request.

wolfCrypt offers the ability to:

1. Create KMS keys,
2. Generate signing key pairs,
3. Validate signing key pairs,
4. Sign messages and
5. Verify messages.

KMS:

1. Initialize ECCSI Key: wc_InitEccsiKey()
2. Make and save or load ECCSI Key:
    - wc_MakeEccsiKey(), wc_ExportEccsiKey(), wc_ExportEccsiPublicKey() or
    - wc_ImportEccsiKey()
3. Wait for request:
    - Receive signing ID from client.
    - Generate signing key pair from ID: wc_MakeEccsiPair()
    - Encode result:
        - For signer, signing key pair: wc_EncodeEccsiPair()
    - Send KPAK and result
4. Free ECCSI Key: wc_FreeEccsiKey()

Client, signer:

1. Initialize ECCSI Key: wc_InitEccsiKey()
2. (When signing pair not cached) Request KPAK and signing pair from KMS
   - Send signing ID to KMS.
   - Receive signing key pair from KMS.
   - Load KMS Public Key: wc_ImportEccsiPublicKey()
   - Decode signing key pair: wc_DecodeEccsiPair()
   - Validate the key pair: wc_ValidateEccsiPair()
3. (If not done above) Load KMS Public Key: wc_ImportEccsiPublicKey()
4. (If not cached) Calculate hash of the ID and PVT: wc_HashEccsiId()
5. For each message:
   - Set Hash of Identity: wc_SetEccsiHash()
   - Sign message: wc_SignEccsiHash()
   - Send hash ID, message and signature to peer.
6. Free ECCSI Key: wc_FreeEccsiKey()

Client, verifier:

1. Receive hash ID, message and signature from signer.
2. Request KPAK (if not cached) and PVT (if not cached) for hash ID from KMS.
3. Receive KPAK (if not cached) and PVT (if not cached) for hash ID from KMS.
4. Initialize ECCSI Key: wc_InitEccsiKey()
5. Load KMS Public Key: wc_ImportEccsiPublicKey()
6. Decode PVT: wc_DecodeEccsiPvtFromSig()
7. Calculate hash of the ID and PVT: wc_HashEccsiId()
8. Set ECCSI key pair: wc_SetEccsiPair()
9. Verify signature of message: wc_VerifyEccsiHash()
10. Free ECCSI Key: wc_FreeEccsiKey()

B.28 Setup ECCSI Key

More...

B.28.1 Functions

	Name
int	**wc_InitEccsiKey**(EccsiKey * key, void * heap, int devId)
int	**wc_InitEccsiKey_ex**(EccsiKey * key, int keySz, int curveId, void * heap, int devId)
void	**wc_FreeEccsiKey**(EccsiKey * key)
int	**wc_MakeEccsiKey**(EccsiKey * key, WC_RNG * rng)
int	**wc_ExportEccsiKey**(EccsiKey * key, byte * data, word32 * sz)
int	**wc_ImportEccsiKey**(EccsiKey * key, const byte * data, word32 sz)
int	**wc_ExportEccsiPrivateKey**(EccsiKey * key, byte * data, word32 * sz)
int	**wc_ImportEccsiPrivateKey**(EccsiKey * key, const byte * data, word32 sz)
int	**wc_ExportEccsiPublicKey**(EccsiKey * key, byte * data, word32 * sz, int raw)

	Name
int	**wc_ImportEccsiPublicKey**(EccsiKey * key, const byte * data, word32 sz, int trusted)
int	**wc_SetEccsiHash**(EccsiKey * key, const byte * hash, byte hashSz)
int	**wc_SetEccsiPair**(EccsiKey * key, const mp_int * ssk, const ecc_point * pvt)

B.28.2 Detailed Description

Operations for establinshing an ECCSI key.

Initialize ECCSI Key before use (wc_InitEccsiKey()).

Initialize ECCSI Key before use (wc_InitEccsiKey_ex()) for use with a curve other than P256.

Either make a new key (wc_MakeEccsiKey()), import an existing key (wc_ImportEccsiKey()) or import existing private key (wc_ImportEccsiPrivateKey()) and public key (wc_ImportEccsiPublicKey()).

Export the key (wc_ExportEccsiKey()) after making a new key for future use.

Export the private key (wc_ExportEccsiPrivateKey()) after making a new key for future use.

Export the public key (wc_ExportEccsiPublicKey()) from KMS to pass to client.

Import the public key (wc_ImportEccsiPublicKey()) into client.

Free the ECCSI Key (wc_FreeEccsiKey()) when finished.

B.28.3 Functions Documentation

```
int wc_InitEccsiKey(
    EccsiKey * key,
    void * heap,
    int devId
)
```

```
int wc_InitEccsiKey_ex(
    EccsiKey * key,
    int keySz,
    int curveId,
    void * heap,
    int devId
)
```

```
void wc_FreeEccsiKey(
    EccsiKey * key
)
```

```
int wc_MakeEccsiKey(
    EccsiKey * key,
    WC_RNG * rng
)
```

```
int wc_ExportEccsiKey(
    EccsiKey * key,
    byte * data,
    word32 * sz
)
```

```
int wc_ImportEccsiKey(
    EccsiKey * key,
    const byte * data,
    word32 sz
)
```

```
int wc_ExportEccsiPrivateKey(
    EccsiKey * key,
    byte * data,
    word32 * sz
)
```

```
int wc_ImportEccsiPrivateKey(
    EccsiKey * key,
    const byte * data,
    word32 sz
)
```

```
int wc_ExportEccsiPublicKey(
    EccsiKey * key,
    byte * data,
    word32 * sz,
    int raw
)
```

```
int wc_ImportEccsiPublicKey(
    EccsiKey * key,
    const byte * data,
    word32 sz,
    int trusted
)



int wc_SetEccsiHash(
    EccsiKey * key,
    const byte * hash,
    byte hashSz
)



int wc_SetEccsiPair(
    EccsiKey * key,
    const mp_int * ssk,
    const ecc_point * pvt
)
```

B.29 Operations for Signing and Verifying with ECCSI Key

B.28.3.12 function wc_SetEccsiPair More...

B.29.1 Functions

	Name
int	**wc_MakeEccsiPair**(EccsiKey * key, WC_RNG * rng, enum wc_HashType hashType, const byte * id, word32 idSz, mp_int * ssk, ecc_point * pvt)
int	**wc_ValidateEccsiPair**(EccsiKey * key, enum wc_HashType hashType, const byte * id, word32 idSz, const mp_int * ssk, ecc_point * pvt, int * valid)
int	**wc_ValidateEccsiPvt**(EccsiKey * key, const ecc_point * pvt, int * valid)
int	**wc_EncodeEccsiPair**(const EccsiKey * key, mp_int * ssk, ecc_point * pvt, byte * data, word32 * sz)
int	**wc_EncodeEccsiSsk**(const EccsiKey * key, mp_int * ssk, byte * data, word32 * sz)
int	**wc_EncodeEccsiPvt**(const EccsiKey * key, ecc_point * pvt, byte * data, word32 * sz, int raw)
int	**wc_DecodeEccsiPair**(const EccsiKey * key, const byte * data, word32 sz, mp_int * ssk, ecc_point * pvt)

	Name
int	**wc_DecodeEccsiSsk**(const EccsiKey * key, const byte * data, word32 sz, mp_int * ssk)
int	**wc_DecodeEccsiPvt**(const EccsiKey * key, const byte * data, word32 sz, ecc_point * pvt)
int	**wc_DecodeEccsiPvtFromSig**(const EccsiKey * key, const byte * sig, word32 sz, ecc_point * pvt)
int	**wc_HashEccsiId**(EccsiKey * key, enum wc_HashType hashType, const byte * id, word32 idSz, ecc_point * pvt, byte * hash, byte * hashSz)
int	**wc_SignEccsiHash**(EccsiKey * key, WC_RNG * rng, enum wc_HashType hashType, const byte * msg, word32 msgSz, byte * sig, word32 * sigSz)
int	**wc_VerifyEccsiHash**(EccsiKey * key, enum wc_HashType hashType, const byte * msg, word32 msgSz, const byte * sig, word32 sigSz, int * verified)

B.29.2 Detailed Description

These operations are for signing and verifying with ECCSI keys.

Make an ECCSI key pair (wc_MakeEccsiPair()) with the signer's ID for use when signing.

Validate the ECCSI key pair (wc_ValidateEccsiPair()) with the signer's ID.

Validate the ECCSI Public Validation Token (PVT) (wc_ValidateEccsiPvt()).

Encode the ECCSI key pair (wc_EncodeEccsiPair()) for transfer to client.

Encode the ECCSI SSK (wc_EncodeEccsiSsk()) for transfer to client.

Encode the ECCSI PVT (wc_EncodeEccsiPvt()) for transfer to verifier.

Decode the ECCSI key pair (wc_DecodeEccsiPair()) on client for signing.

Decode the ECCSI SSK (wc_DecodeEccsiSsk()) on client for signing.

Decode the ECCSI PVT (wc_DecodeEccsiPvt()) on client for signing.

Decode the ECCSI PVT from the signature (wc_DecodeEccsiPvtFromSig()) on client for verifying.

Calculate hash of the ID (wc_HashEccsiId()) for signing/verifying using ID and Public Validation Token (PVT).

Sign (wc_SignEccsiHash()) a message with the hash of the ID and the Secret Signing Key (SSK) and Public Validation Token (PVT).

Verify (wc_VerifyEccsiHash()) a message with the hash of the signer's ID.

B.29.3 Functions Documentation

```
int wc_MakeEccsiPair(
    EccsiKey * key,
    WC_RNG * rng,
    enum wc_HashType hashType,
    const byte * id,
    word32 idSz,
```

```
    mp_int * ssk,
    ecc_point * pvt
)



int wc_ValidateEccsiPair(
    EccsiKey * key,
    enum wc_HashType hashType,
    const byte * id,
    word32 idSz,
    const mp_int * ssk,
    ecc_point * pvt,
    int * valid
)



int wc_ValidateEccsiPvt(
    EccsiKey * key,
    const ecc_point * pvt,
    int * valid
)



int wc_EncodeEccsiPair(
    const EccsiKey * key,
    mp_int * ssk,
    ecc_point * pvt,
    byte * data,
    word32 * sz
)



int wc_EncodeEccsiSsk(
    const EccsiKey * key,
    mp_int * ssk,
    byte * data,
    word32 * sz
)



int wc_EncodeEccsiPvt(
    const EccsiKey * key,
    ecc_point * pvt,
    byte * data,
    word32 * sz,
    int raw
)
```

```
int wc_DecodeEccsiPair(
    const EccsiKey * key,
    const byte * data,
    word32 sz,
    mp_int * ssk,
    ecc_point * pvt
)
```

```
int wc_DecodeEccsiSsk(
    const EccsiKey * key,
    const byte * data,
    word32 sz,
    mp_int * ssk
)
```

```
int wc_DecodeEccsiPvt(
    const EccsiKey * key,
    const byte * data,
    word32 sz,
    ecc_point * pvt
)
```

```
int wc_DecodeEccsiPvtFromSig(
    const EccsiKey * key,
    const byte * sig,
    word32 sz,
    ecc_point * pvt
)
```

```
int wc_HashEccsiId(
    EccsiKey * key,
    enum wc_HashType hashType,
    const byte * id,
    word32 idSz,
    ecc_point * pvt,
    byte * hash,
    byte * hashSz
)
```

```
int wc_SignEccsiHash(
    EccsiKey * key,
    WC_RNG * rng,
    enum wc_HashType hashType,
    const byte * msg,
    word32 msgSz,
    byte * sig,
    word32 * sigSz
)



int wc_VerifyEccsiHash(
    EccsiKey * key,
    enum wc_HashType hashType,
    const byte * msg,
    word32 msgSz,
    const byte * sig,
    word32 sigSz,
    int * verified
)
```

B.30 Overview of SAKKE Key

B.29.3.13 function wc_VerifyEccsiHash More...

B.30.1 Detailed Description

SAKKE (Sakai-Kasahara Key Encryption) is specified in RFC 6508 (https://tools.ietf.org/html/rfc6508).

SAKKE is used to transfer a secret to a peer using Identity Based cryptography.

The Key Management Service (KMS) is responsible for issuing Receiver Secret Keys (RSKs). Data up to $(2^{hashlen)}$hashlen bytes of data can be transferred.

The sender must know the identity of the receiver and the KMS Public Key.

The receiver must have obtained a Receiver Secret Key (RSK) for the identity from a KMS in order to derive the secret.

KMS:

1. Initialize SAKKE Key: wc_InitSakkeKey()
2. Make and save or load SAKKE Key:
    - wc_MakeSakkeKey(), wc_ExportSakkeKey(), wc_ExportSakkePublicKey() or
    - wc_ImportSakkeKey()
3. Wait for request:
    - Make an RSK base on ID for the client: wc_MakeSakkeRsk()
    - Encode RSK for transfer to client: wc_EncodeSakkeRsk()
4. Free SAKKE Key: wc_FreeSakkeKey()

Key Exchange, Peer A:

1. Initialize SAKKE Key: wc_InitSakkeKey()
2. Load KMS Public Key: wc_ImportSakkePublicKey()

3. Generate a random SSV: wc_GenerateSakkeSSV()
4. Set the identity of Peer B: wc_SetSakkeIdentity()
5. Make an encapsulated SSV and auth data: wc_MakeSakkeEncapsulatedSSV()
6. Send encapsulated data to Peer B
7. Free SAKKE Key: wc_FreeSakkeKey()

Key Exchange, Peer B:

1. Receive encapsulated data.
2. Initialize SAKKE Key: wc_InitSakkeKey()
3. Load KMS Public Key: wc_ImportSakkePublicKey()
4. Decode RSK transferred from KMS or stored locally: wc_DecodeSakkeRsk()
5. [Optional] Validate RSK before first use: wc_ValidateSakkeRsk()
6. Set the identity: wc_SetSakkeIdentity()
7. Set the RSK and, optionally precomputation table: wc_SetSakkeRsk()
8. Derive SSV with auth data: wc_DeriveSakkeSSV()
9. Free SAKKE Key: wc_FreeSakkeKey()

Transfer secret, Peer A:

1. Initialize SAKKE Key: wc_InitSakkeKey()
2. Load KMS Public Key: wc_ImportSakkePublicKey()
3. Set the identity of Peer B: wc_SetSakkeIdentity()
4. Make an encapsulation of the SSV and auth data: wc_MakeSakkeEncapsulatedSSV()
5. Send encapsulated data to Peer B
6. Free SAKKE Key: wc_FreeSakkeKey()

Transfer secret, Peer B:

1. Initialize SAKKE Key: wc_InitSakkeKey()
2. Load KMS Public Key: wc_ImportSakkePublicKey()
3. Decode RSK transferred from KMS or stored locally: wc_DecodeSakkeRsk()
4. [Optional] Validate RSK before first use: wc_ValidateSakkeRsk()
5. Receive encapsulated data.
6. Set the identity: wc_SetSakkeIdentity()
7. Set the RSK and, optionally precomputation table: wc_SetSakkeRsk()
8. Derive SSV and auth data: wc_DeriveSakkeSSV()
9. Free SAKKE Key: wc_FreeSakkeKey()

B.31 Setup SAKKE Key

More...

B.31.1 Functions

	Name
int	**wc_InitSakkeKey**(SakkeKey * key, void * heap, int devId)
int	**wc_InitSakkeKey_ex**(SakkeKey * key, int keySize, int curveId, void * heap, int devId)
void	**wc_FreeSakkeKey**(SakkeKey * key)
int	**wc_MakeSakkeKey**(SakkeKey * key, WC_RNG * rng)
int	**wc_MakeSakkePublicKey**(SakkeKey * key, ecc_point * pub)

	Name
int	**wc_ExportSakkeKey**(SakkeKey * key, byte * data, word32 * sz)
int	**wc_ImportSakkeKey**(SakkeKey * key, const byte * data, word32 sz)
int	**wc_ExportSakkePrivateKey**(SakkeKey * key, byte * data, word32 * sz)
int	**wc_ImportSakkePrivateKey**(SakkeKey * key, const byte * data, word32 sz)
int	**wc_ExportSakkePublicKey**(SakkeKey * key, byte * data, word32 * sz, int raw)
int	**wc_ImportSakkePublicKey**(SakkeKey * key, const byte * data, word32 sz, int trusted)
int	**wc_SetSakkeIdentity**(SakkeKey * key, const byte * id, word16 idSz)

B.31.2 Detailed Description

Operations for establishing a SAKKE key.

Initialization SAKKE Key before use (wc_InitSakkeKey() or wc_InitSakkeKey_ex()).

Either make a new key (wc_MakeSakkeKey()) or import an existing key (wc_ImportSakkeKey()).

Export the key (wc_ExportSakkeKey()) after making a new key for future use.

If only the private part of the KMS SAKKE Key is available, make the public key (wc_MakeSakkePublicKey()).

Export the private key (wc_ExportSakkePrivateKey()) from KMS from storage.

Import the private key (wc_ImportSakkePrivateKey()) into KMS from storage.

Export the public key (wc_ExportSakkePublicKey()) from KMS to pass to client.

Import the public key (wc_ImportSakkePublicKey()) into client.

Set the identity to use (wc_SetSakkeIdentity()) into client.

Free the SAKKE Key (wc_FreeSakkeKey()) when finished.

B.31.3 Functions Documentation

```
int wc_InitSakkeKey(
    SakkeKey * key,
    void * heap,
    int devId
)
```

```
int wc_InitSakkeKey_ex(
    SakkeKey * key,
    int keySize,
    int curveId,
    void * heap,
    int devId
)
```

```
void wc_FreeSakkeKey(
    SakkeKey * key
)
```

```
int wc_MakeSakkeKey(
    SakkeKey * key,
    WC_RNG * rng
)
```

```
int wc_MakeSakkePublicKey(
    SakkeKey * key,
    ecc_point * pub
)
```

```
int wc_ExportSakkeKey(
    SakkeKey * key,
    byte * data,
    word32 * sz
)
```

```
int wc_ImportSakkeKey(
    SakkeKey * key,
    const byte * data,
    word32 sz
)
```

```
int wc_ExportSakkePrivateKey(
    SakkeKey * key,
    byte * data,
    word32 * sz
)
```

```
int wc_ImportSakkePrivateKey(
    SakkeKey * key,
    const byte * data,
    word32 sz
)
```

```
int wc_ExportSakkePublicKey(
    SakkeKey * key,
    byte * data,
    word32 * sz,
    int raw
)
```

```
int wc_ImportSakkePublicKey(
    SakkeKey * key,
    const byte * data,
    word32 sz,
    int trusted
)
```

```
int wc_SetSakkeIdentity(
    SakkeKey * key,
    const byte * id,
    word16 idSz
)
```

B.32 Operations on/with SAKKE RSK

B.31.3.12 function wc_SetSakkeIdentity More...

B.32.1 Functions

	Name
int	**wc_MakeSakkeRsk**(SakkeKey * key, const byte * id, word16 idSz, ecc_point * rsk)
int	**wc_ValidateSakkeRsk**(SakkeKey * key, const byte * id, word16 idSz, ecc_point * rsk, int * valid)
int	**wc_GenerateSakkeRskTable**(const SakkeKey * key, const ecc_point * rsk, byte * table, word32 * len)
int	**wc_EncodeSakkeRsk**(const SakkeKey * key, ecc_point * rsk, byte * out, word32 * sz, int raw)
int	**wc_DecodeSakkeRsk**(const SakkeKey * key, const byte * data, word32 sz, ecc_point * rsk)
int	**wc_ImportSakkeRsk**(SakkeKey * key, const byte * data, word32 sz)
int	**wc_SetSakkeRsk**(SakkeKey * key, const ecc_point * rsk, byte * table, word32 len)

B.32.2 Detailed Description

These operations make, validate, encode and decode a Receiver Secret Key (RSK).

An RSK is required to derive an SSV (see wc_DeriveSakkeSSV()).

On the KMS, make an RSK (wc_MakeSakkeRsk()) from the client's ID.

On the client, validate the RSK (wc_ValidateSakkeRsk()) with the ID.

Encode the RSK (wc_EncodeSakkeRsk()) to pass to client or for storage.

Decode the RSK (wc_DecodeSakkeRsk()) on the client when needed.

Import the RSK (wc_ImportSakkeRsk()) on the client when needed.

Set the RSK and, optionally, a pre-computation table (wc_SetSakkeRsk()) on the client when needed.

B.32.3 Functions Documentation

```
int wc_MakeSakkeRsk(
    SakkeKey * key,
    const byte * id,
    word16 idSz,
    ecc_point * rsk
)
```

```
int wc_ValidateSakkeRsk(
    SakkeKey * key,
    const byte * id,
    word16 idSz,
    ecc_point * rsk,
    int * valid
)
```

```
int wc_GenerateSakkeRskTable(
    const SakkeKey * key,
    const ecc_point * rsk,
    byte * table,
    word32 * len
)
```

```
int wc_EncodeSakkeRsk(
    const SakkeKey * key,
    ecc_point * rsk,
    byte * out,
    word32 * sz,
    int raw
)
```

```
int wc_DecodeSakkeRsk(
    const SakkeKey * key,
    const byte * data,
    word32 sz,
    ecc_point * rsk
)
```

```
int wc_ImportSakkeRsk(
    SakkeKey * key,
    const byte * data,
    word32 sz
)
```

```
int wc_SetSakkeRsk(
    SakkeKey * key,
    const ecc_point * rsk,
    byte * table,
    word32 len
)
```

B.33 Operations using SAKKE Key

B.32.3.7 function wc_SetSakkeRsk More...

B.33.1 Functions

	Name
int	**wc_GetSakkeAuthSize**(SakkeKey * key, word16 * authSz)
int	**wc_MakeSakkePointI**(SakkeKey * key, const byte * id, word16 idSz)
int	**wc_GetSakkePointI**(SakkeKey * key, byte * data, word32 * sz)
int	**wc_SetSakkePointI**(SakkeKey * key, const byte * id, word16 idSz, const byte * data, word32 sz)
int	**wc_GenerateSakkePointITable**(SakkeKey * key, byte * table, word32 * len)
int	**wc_SetSakkePointITable**(SakkeKey * key, byte * table, word32 len)
int	**wc_ClearSakkePointITable**(SakkeKey * key)
int	**wc_MakeSakkeEncapsulatedSSV**(SakkeKey * key, enum wc_HashType hashType, byte * ssv, word16 ssvSz, byte * auth, word16 * authSz)
int	**wc_GenerateSakkeSSV**(SakkeKey * key, WC_RNG * rng, byte * ssv, word16 * ssvSz)

	Name
int	**wc_DeriveSakkeSSV**(SakkeKey * key, enum wc_HashType hashType, byte * ssv, word16 ssvSz, const byte * auth, word16 authSz)

B.33.2 Detailed Description

These operations transfer a Shared Secret Value (SSV) from one client to another. The SSV may be randomly generated.

Calculate the size of the authentication data (wc_GetSakkeAuthSize()) to determine where the SSV starts in a buffer.

Make the intermediate point I (wc_MakeSakkePointI()) to speed making an encapsulated and deriving SSV.

Get intermediate point I (wc_GetSakkePointI()) for storage.

Set intermediate point I (wc_SetSakkePointI()) from storage.

Generate a pre-computation table for intermediate point I (wc_GenerateSakkePointITable()) to further enhance performance. Store as necessary.

Set the pre-computation table for intermediate point I (wc_SetSakkePointITable()) to further enhance performance.

Clear the pre-computation table for intermediate point I (wc_ClearSakkePointITable()) to remove reference to external table pointer.

Make an encapsulated SSV (wc_MakeSakkeEncapsulatedSSV()) to share with another client. Data in SSV is modified.

Generate a random SSV (wc_GenerateSakkeSSV()) for key exchange.

Derive the SSV, (wc_DeriveSakkeSSV()) on the recipient from the encapsulated SSV.

B.33.3 Functions Documentation

```
int wc_GetSakkeAuthSize(
    SakkeKey * key,
    word16 * authSz
)
```

```
int wc_MakeSakkePointI(
    SakkeKey * key,
    const byte * id,
    word16 idSz
)
```

```
int wc_GetSakkePointI(
    SakkeKey * key,
    byte * data,
    word32 * sz
)
```

```
int wc_SetSakkePointI(
    SakkeKey * key,
    const byte * id,
    word16 idSz,
    const byte * data,
    word32 sz
)
```

```
int wc_GenerateSakkePointITable(
    SakkeKey * key,
    byte * table,
    word32 * len
)
```

```
int wc_SetSakkePointITable(
    SakkeKey * key,
    byte * table,
    word32 len
)
```

```
int wc_ClearSakkePointITable(
    SakkeKey * key
)
```

```
int wc_MakeSakkeEncapsulatedSSV(
    SakkeKey * key,
    enum wc_HashType hashType,
    byte * ssv,
    word16 ssvSz,
    byte * auth,
    word16 * authSz
)
```

```
int wc_GenerateSakkeSSV(
    SakkeKey * key,
    WC_RNG * rng,
    byte * ssv,
    word16 * ssvSz
)
```

```
int wc_DeriveSakkeSSV(
    SakkeKey * key,
    enum wc_HashType hashType,
    byte * ssv,
    word16 ssvSz,
    const byte * auth,
    word16 authSz
)
```

B.34 Algorithms - HMAC

B.33.3.10 function wc_DeriveSakkeSSV

B.34.1 Functions

	Name
int	**wc_HmacSetKey**(Hmac * hmac, int type, const byte * key, word32 keySz)This function initializes an Hmac object, setting its encryption type, key and HMAC length.
int	**wc_HmacUpdate**(Hmac * hmac, const byte * in, word32 sz)This function updates the message to authenticate using HMAC. It should be called after the Hmac object has been initialized with wc_HmacSetKey. This function may be called multiple times to update the message to hash. After calling wc_HmacUpdate as desired, one should call wc_HmacFinal to obtain the final authenticated message tag.
int	**wc_HmacFinal**(Hmac * hmac, byte * out)This function computes the final hash of an Hmac object's message.
int	**wolfSSL_GetHmacMaxSize**(void)This function returns the largest HMAC digest size available based on the configured cipher suites.
int	**wc_HKDF**(int type, const byte * inKey, word32 inKeySz, const byte * salt, word32 saltSz, const byte * info, word32 infoSz, byte * out, word32 outSz)This function provides access to a HMAC Key Derivation Function (HKDF). It utilizes HMAC to convert inKey, with an optional salt and optional info into a derived key, which it stores in out. The hash type defaults to MD5 if 0 or NULL is given.

	Name
int	**wc_HKDF_Extract**(int type, const byte * salt, word32 saltSz, const byte * inKey, word32 inKeySz, byte * out)This function provides access to a HMAC Key Derivation Function (HKDF). It utilizes HMAC to convert inKey, with an optional salt into a derived key, which it stores in out. The hash type defaults to MD5 if 0 or NULL is given.
int	**wc_HKDF_Extract_ex**(int type, const byte * salt, word32 saltSz, const byte * inKey, word32 inKeySz, byte * out, void * heap, int devId)This function provides access to a HMAC Key Derivation Function (HKDF). It utilizes HMAC to convert inKey, with an optional salt into a derived key, which it stores in out. The hash type defaults to MD5 if 0 or NULL is given. This is the _ex version adding heap hint and device identifier.
int	**wc_HKDF_Expand**(int type, const byte * inKey, word32 inKeySz, const byte * info, word32 infoSz, byte * out, word32 outSz)This function provides access to a HMAC Key Derivation Function (HKDF). It utilizes HMAC to convert inKey, with optional info into a derived key, which it stores in out. The hash type defaults to MD5 if 0 or NULL is given.
int	**wc_HKDF_Expand_ex**(int type, const byte * inKey, word32 inKeySz, const byte * info, word32 infoSz, byte * out, word32 outSz, void * heap, int devId)This function provides access to a HMAC Key Derivation Function (HKDF). It utilizes HMAC to convert inKey, with optional info into a derived key, which it stores in out. The hash type defaults to MD5 if 0 or NULL is given. This is the _ex version adding heap hint and device identifier.
int	**wc_Tls13_HKDF_Extract**(byte * prk, const byte * salt, word32 saltLen, byte * ikm, word32 ikmLen, int digest)This function provides access to RFC 5869 HMAC_based Extract_and_Expand Key Derivation Function (HKDF) for TLS v1.3 key derivation.
int	**wc_Tls13_HKDF_Extract_ex**(byte * prk, const byte * salt, word32 saltLen, byte * ikm, word32 ikmLen, int digest, void * heap, int devId)This function provides access to RFC 5869 HMAC_based Extract_and_Expand Key Derivation Function (HKDF) for TLS v1.3 key derivation. This is the _ex version adding heap hint and device identifier.

	Name
int	**wc_Tls13_HKDF_Expand_Label_ex**(byte * okm, word32 okmLen, const byte * prk, word32 prkLen, const byte * protocol, word32 protocolLen, const byte * label, word32 labelLen, const byte * info, word32 infoLen, int digest, void * heap, int devId)Expand data using HMAC, salt and label and info. TLS v1.3 defines this function for key derivation. This is the _ex version adding heap hint and device identifier.
int	**wc_Tls13_HKDF_Expand_Label**(byte * okm, word32 okmLen, const byte * prk, word32 prkLen, const byte * protocol, word32 protocolLen, const byte * label, word32 labelLen, const byte * info, word32 infoLen, int digest)Expand data using HMAC, salt and label and info. TLS v1.3 defines this function for key derivation.
int	**wc_Tls13_HKDF_Expand_Label_Alloc**(byte * okm, word32 okmLen, const byte * prk, word32 prkLen, const byte * protocol, word32 protocolLen, const byte * label, word32 labelLen, const byte * info, word32 infoLen, int digest, void * heap)This functions is very similar to wc_Tls13_HKDF_Expand_Label(), but it allocates memory if the stack space usually used isn't enough. Expand data using HMAC, salt and label and info. TLS v1.3 defines this function for key derivation. This is the _ex version adding heap hint and device identifier.

B.34.2 Functions Documentation

```
int wc_HmacSetKey(
    Hmac * hmac,
    int type,
    const byte * key,
    word32 keySz
)
```

This function initializes an Hmac object, setting its encryption type, key and HMAC length.

Parameters:

- **hmac** pointer to the Hmac object to initialize
- **type** type specifying which encryption method the Hmac object should use. Valid options are: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512
- **key** pointer to a buffer containing the key with which to initialize the Hmac object
- **length** length of the key

See:

- wc_HmacUpdate
- wc_HmacFinal

Return:

- 0 Returned on successfully initializing the Hmac object
- BAD_FUNC_ARG Returned if the input type is invalid (see type param)
- MEMORY_E Returned if there is an error allocating memory for the structure to use for hashing
- HMAC_MIN_KEYLEN_E Returned when using a FIPS implementation and the key length specified is shorter than the minimum acceptable FIPS standard of 14 bytes

Example

```
Hmac hmac;
byte key[] = { // initialize with key to use for encryption };
if (wc_HmacSetKey(&hmac, WC_MD5, key, sizeof(key)) != 0) {
    // error initializing Hmac object
}
```

```
int wc_HmacUpdate(
    Hmac * hmac,
    const byte * in,
    word32 sz
)
```

This function updates the message to authenticate using HMAC. It should be called after the Hmac object has been initialized with wc_HmacSetKey. This function may be called multiple times to update the message to hash. After calling wc_HmacUpdate as desired, one should call wc_HmacFinal to obtain the final authenticated message tag.

Parameters:

- **hmac** pointer to the Hmac object for which to update the message
- **msg** pointer to the buffer containing the message to append
- **length** length of the message to append

See:

- wc_HmacSetKey
- wc_HmacFinal

Return:

- 0 Returned on successfully updating the message to authenticate
- MEMORY_E Returned if there is an error allocating memory for use with a hashing algorithm

Example

```
Hmac hmac;
byte msg[] = { // initialize with message to authenticate };
byte msg2[] = { // initialize with second half of message };
// initialize hmac
if( wc_HmacUpdate(&hmac, msg, sizeof(msg)) != 0) {
    // error updating message
}
if( wc_HmacUpdate(&hmac, msg2, sizeof(msg)) != 0) {
    // error updating with second message
}
```

```
int wc_HmacFinal(
    Hmac * hmac,
    byte * out
)
```

This function computes the final hash of an Hmac object's message.

Parameters:

- **hmac** pointer to the Hmac object for which to calculate the final hash
- **hash** pointer to the buffer in which to store the final hash. Should have room available as required by the hashing algorithm chosen

See:

- wc_HmacSetKey
- wc_HmacUpdate

Return:

- 0 Returned on successfully computing the final hash
- MEMORY_E Returned if there is an error allocating memory for use with a hashing algorithm

Example

```
Hmac hmac;
byte hash[MD5_DIGEST_SIZE];
// initialize hmac with MD5 as type
// wc_HmacUpdate() with messages

if (wc_HmacFinal(&hmac, hash) != 0) {
    // error computing hash
}
```

```
int wolfSSL_GetHmacMaxSize(
    void
)
```

This function returns the largest HMAC digest size available based on the configured cipher suites.

Parameters:

- **none** No parameters.

See: none

Return: Success Returns the largest HMAC digest size available based on the configured cipher suites

Example

```
int maxDigestSz = wolfSSL_GetHmacMaxSize();
```

```
int wc_HKDF(
    int type,
    const byte * inKey,
    word32 inKeySz,
    const byte * salt,
    word32 saltSz,
    const byte * info,
    word32 infoSz,
    byte * out,
    word32 outSz
)
```

This function provides access to a HMAC Key Derivation Function (HKDF). It utilizes HMAC to convert inKey, with an optional salt and optional info into a derived key, which it stores in out. The hash type defaults to MD5 if 0 or NULL is given.

Parameters:

- **type** hash type to use for the HKDF. Valid types are: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512
- **inKey** pointer to the buffer containing the key to use for KDF
- **inKeySz** length of the input key
- **salt** pointer to a buffer containing an optional salt. Use NULL instead if not using a salt
- **saltSz** length of the salt. Use 0 if not using a salt
- **info** pointer to a buffer containing optional additional info. Use NULL if not appending extra info
- **infoSz** length of additional info. Use 0 if not using additional info
- **out** pointer to the buffer in which to store the derived key
- **outSz** space available in the output buffer to store the generated key

See: wc_HmacSetKey

Return:

- 0 Returned upon successfully generating a key with the given inputs
- BAD_FUNC_ARG Returned if an invalid hash type is given (see type param)
- MEMORY_E Returned if there is an error allocating memory
- HMAC_MIN_KEYLEN_E May be returned when using a FIPS implementation and the key length specified is shorter than the minimum acceptable FIPS standard

The HMAC configure option is –enable-hmac (on by default) or if building sources directly HAVE_HKDF

Example

```
byte key[] = { // initialize with key };
byte salt[] = { // initialize with salt };
byte derivedKey[MAX_DIGEST_SIZE];

int ret = wc_HKDF(WC_SHA512, key, sizeof(key), salt, sizeof(salt),
NULL, 0, derivedKey, sizeof(derivedKey));
if ( ret != 0 ) {
    // error generating derived key
}
```

```
int wc_HKDF_Extract(
    int type,
    const byte * salt,
    word32 saltSz,
    const byte * inKey,
    word32 inKeySz,
    byte * out
)
```

This function provides access to a HMAC Key Derivation Function (HKDF). It utilizes HMAC to convert inKey, with an optional salt into a derived key, which it stores in out. The hash type defaults to MD5 if 0 or NULL is given.

Parameters:

- **type** hash type to use for the HKDF. Valid types are: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512
- **salt** pointer to a buffer containing an optional salt. Use NULL instead if not using a salt
- **saltSz** length of the salt. Use 0 if not using a salt
- **inKey** pointer to the buffer containing the key to use for KDF
- **inKeySz** length of the input key
- **out** pointer to the buffer in which to store the derived key

See:

- wc_HKDF
- wc_HKDF_Extract_ex
- wc_HKDF_Expand
- wc_HKDF_Expand_ex

Return:

- 0 Returned upon successfully generating a key with the given inputs
- BAD_FUNC_ARG Returned if an invalid hash type is given (see type param)
- MEMORY_E Returned if there is an error allocating memory

- HMAC_MIN_KEYLEN_E May be returned when using a FIPS implementation and the key length specified is shorter than the minimum acceptable FIPS standard

The HMAC configure option is –enable-hmac (on by default) or if building sources directly HAVE_HKDF

Example

```
byte key[] = { // initialize with key };
byte salt[] = { // initialize with salt };
byte derivedKey[MAX_DIGEST_SIZE];

int ret = wc_HKDF_Extract(WC_SHA512, salt, sizeof(salt), key, sizeof(key),
    derivedKey);
if ( ret != 0 ) {
    // error generating derived key
}
```

```
int wc_HKDF_Extract_ex(
    int type,
    const byte * salt,
    word32 saltSz,
    const byte * inKey,
    word32 inKeySz,
    byte * out,
    void * heap,
    int devId
)
```

This function provides access to a HMAC Key Derivation Function (HKDF). It utilizes HMAC to convert inKey, with an optional salt into a derived key, which it stores in out. The hash type defaults to MD5 if 0 or NULL is given. This is the _ex version adding heap hint and device identifier.

Parameters:

- **type** hash type to use for the HKDF. Valid types are: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512
- **salt** pointer to a buffer containing an optional salt. Use NULL instead if not using a salt
- **saltSz** length of the salt. Use 0 if not using a salt
- **inKey** pointer to the buffer containing the key to use for KDF
- **inKeySz** length of the input key
- **out** pointer to the buffer in which to store the derived key. Must be digest length per hash 'type'. See wc_HmacSizeByType()
- **heap** heap hint to use for memory. Can be NULL
- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID (-2) if not used

See:

- wc_HKDF
- wc_HKDF_Extract
- wc_HKDF_Expand
- wc_HKDF_Expand_ex

- wc_HmacSizeByType

Return:

- 0 Returned upon successfully generating a key with the given inputs
- BAD_FUNC_ARG Returned if an invalid hash type is given (see type param)
- MEMORY_E Returned if there is an error allocating memory
- HMAC_MIN_KEYLEN_E May be returned when using a FIPS implementation and the key length specified is shorter than the minimum acceptable FIPS standard
- WC_PENDING_E May be returned in a WOLF_CRYPTO_CB build when the registered crypto callback device has taken the request but not yet finished it. The caller must re-invoke with identical arguments until the result is no longer WC_PENDING_E; HKDF has no WC_ASYNC_DEV, so this is a poll and not a wc_AsyncWait(). In a WOLFSSL_ASYNC_CRYPT build the TLS 1.3 key schedule resumes a pending HKDF request by re-invoking the callback with identical arguments; without WOLFSSL_ASYNC_CRYPT a device also used for TLS 1.3 must complete HKDF requests synchronously. wc_HKDF_ex() follows the same contract and re-issues its extract step on every retry, so a device that pends must serve a repeated identical request from its completed result. There is no request handle: a device should key its completion tracking on the output pointer plus the argument tuple, and one that cannot correlate a retry that way must not pend.

The HMAC configure option is –enable-hmac (on by default) or if building sources directly HAVE_HKDF

Example

```
byte key[] = { // initialize with key };
byte salt[] = { // initialize with salt };
byte derivedKey[MAX_DIGEST_SIZE];

int ret = wc_HKDF_Extract_ex(WC_SHA512, salt, sizeof(salt), key, sizeof(key),
    derivedKey, NULL, INVALID_DEVID);
if ( ret != 0 ) {
    // error generating derived key
}
```

```
int wc_HKDF_Expand(
    int type,
    const byte * inKey,
    word32 inKeySz,
    const byte * info,
    word32 infoSz,
    byte * out,
    word32 outSz
)
```

This function provides access to a HMAC Key Derivation Function (HKDF). It utilizes HMAC to convert inKey, with optional info into a derived key, which it stores in out. The hash type defaults to MD5 if 0 or NULL is given.

Parameters:

- **type** hash type to use for the HKDF. Valid types are: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512
- **inKey** pointer to the buffer containing the key to use for KDF
- **inKeySz** length of the input key
- **info** pointer to a buffer containing optional additional info. Use NULL if not appending extra info
- **infoSz** length of additional info. Use 0 if not using additional info
- **out** pointer to the buffer in which to store the derived key
- **outSz** space available in the output buffer to store the generated key

See:

- wc_HKDF
- wc_HKDF_Extract
- wc_HKDF_Extract_ex
- wc_HKDF_Expand_ex

Return:

- 0 Returned upon successfully generating a key with the given inputs
- BAD_FUNC_ARG Returned if an invalid hash type is given (see type param)
- MEMORY_E Returned if there is an error allocating memory
- HMAC_MIN_KEYLEN_E May be returned when using a FIPS implementation and the key length specified is shorter than the minimum acceptable FIPS standard

The HMAC configure option is –enable-hmac (on by default) or if building sources directly HAVE_HKDF

Example

```
byte key[] = { // initialize with key };
byte salt[] = { // initialize with salt };
byte derivedKey[MAX_DIGEST_SIZE];

int ret = wc_HKDF_Expand(WC_SHA512, key, sizeof(key), NULL, 0,
    derivedKey, sizeof(derivedKey));
if ( ret != 0 ) {
    // error generating derived key
}
```

```
int wc_HKDF_Expand_ex(
    int type,
    const byte * inKey,
    word32 inKeySz,
    const byte * info,
    word32 infoSz,
    byte * out,
    word32 outSz,
    void * heap,
    int devId
)
```

This function provides access to a HMAC Key Derivation Function (HKDF). It utilizes HMAC to convert inKey, with optional info into a derived key, which it stores in out. The hash type defaults to MD5 if 0 or NULL is given. This is the _ex version adding heap hint and device identifier.

Parameters:

- **type** hash type to use for the HKDF. Valid types are: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512
- **inKey** pointer to the buffer containing the key to use for KDF
- **inKeySz** length of the input key
- **info** pointer to a buffer containing optional additional info. Use NULL if not appending extra info
- **infoSz** length of additional info. Use 0 if not using additional info
- **out** pointer to the buffer in which to store the derived key
- **outSz** space available in the output buffer to store the generated key
- **heap** heap hint to use for memory. Can be NULL
- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID (-2) if not used

See:

- wc_HKDF
- wc_HKDF_Extract
- wc_HKDF_Extract_ex
- wc_HKDF_Expand

Return:

- 0 Returned upon successfully generating a key with the given inputs
- BAD_FUNC_ARG Returned if an invalid hash type is given (see type param)
- MEMORY_E Returned if there is an error allocating memory
- HMAC_MIN_KEYLEN_E May be returned when using a FIPS implementation and the key length specified is shorter than the minimum acceptable FIPS standard
- WC_PENDING_E May be returned in a WOLF_CRYPTO_CB build when the registered crypto callback device has taken the request but not yet finished it. The caller must re-invoke with identical arguments until the result is no longer WC_PENDING_E; HKDF has no WC_ASYNC_DEV, so this is a poll and not a wc_AsyncWait(). In a WOLFSSL_ASYNC_CRYPT build the TLS 1.3 key schedule resumes a pending HKDF request by re-invoking the callback with identical arguments; without WOLFSSL_ASYNC_CRYPT a device also used for TLS 1.3 must complete HKDF requests synchronously. wc_HKDF_ex() follows the same contract and re-issues its extract step on every retry, so a device that pends must serve a repeated identical request from its completed result. There is no request handle: a device should key its completion tracking on the output pointer plus the argument tuple, and one that cannot correlate a retry that way must not pend.

The HMAC configure option is –enable-hmac (on by default) or if building sources directly HAVE_HKDF

Example

```
byte key[] = { // initialize with key };
byte salt[] = { // initialize with salt };
byte derivedKey[MAX_DIGEST_SIZE];

int ret = wc_HKDF_Expand_ex(WC_SHA512, key, sizeof(key), NULL, 0,
    derivedKey, sizeof(derivedKey), NULL, INVALID_DEVID);
if ( ret != 0 ) {
```

```
    // error generating derived key
}



int wc_Tls13_HKDF_Extract(
    byte * prk,
    const byte * salt,
    word32 saltLen,
    byte * ikm,
    word32 ikmLen,
    int digest
)
```

This function provides access to RFC 5869 HMAC-based Extract-and-Expand Key Derivation Function (HKDF) for TLS v1.3 key derivation.

Parameters:

- **prk** Generated pseudorandom key
- **salt** salt. May be NULL, in which case saltLen is ignored
- **saltLen** length of the salt
- **ikm** input keying material. May be NULL when ikmLen is 0
- **ikmLen** length of the input keying material. 0 substitutes a digest length zeroed IKM per RFC 8446, not an empty IKM
- **digest** hash type to use for the HKDF. Valid types are: WC_SHA256, WC_SHA384, WC_SHA512 or WC_SM3

See:

- wc_HKDF
- wc_HKDF_Extract
- wc_HKDF_Extract_ex
- wc_HKDF_Expand
- wc_Tls13_HKDF_Extract_ex

Return:

- 0 Returned upon successfully generating a key with the given inputs
- BAD_FUNC_ARG Returned if an invalid hash type is given (see digest param), if prk is NULL, or if ikm is NULL and ikmLen is not 0
- MEMORY_E Returned if there is an error allocating memory
- HMAC_MIN_KEYLEN_E May be returned when using a FIPS implementation and the key length specified is shorter than the minimum acceptable FIPS standard

Example

```
byte ikm[] = { // initialize with input keying material };
byte salt[] = { // initialize with optional salt };
byte prk[MAX_DIGEST_SIZE];

int ret = wc_Tls13_HKDF_Extract(prk, salt, sizeof(salt), ikm, sizeof(ikm),
```

```
    WC_SHA512);
if ( ret != 0 ) {
    // error generating derived key
}
```

```
int wc_Tls13_HKDF_Extract_ex(
    byte * prk,
    const byte * salt,
    word32 saltLen,
    byte * ikm,
    word32 ikmLen,
    int digest,
    void * heap,
    int devId
)
```

This function provides access to RFC 5869 HMAC-based Extract-and-Expand Key Derivation Function (HKDF) for TLS v1.3 key derivation. This is the _ex version adding heap hint and device identifier.

Parameters:

- **prk** Generated pseudorandom key
- **salt** Salt. May be NULL; saltLen is then ignored unless a crypto callback handles the request
- **saltLen** Length of the salt
- **ikm** Input keying material. May be NULL when ikmLen is 0
- **ikmLen** Length of the input keying material. 0 substitutes a digest length zeroed IKM per RFC 8446, not an empty IKM
- **digest** Hash type to use for the HKDF. Valid types are: WC_SHA256, WC_SHA384, WC_SHA512 or WC_SM3
- **heap** Heap hint to use for memory. Can be NULL
- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID (-2) if not used

See:

- wc_HKDF
- wc_HKDF_Extract
- wc_HKDF_Extract_ex
- wc_HKDF_Expand
- wc_Tls13_HKDF_Extract

Return:

- 0 Returned upon successfully generating a key with the given inputs
- BAD_FUNC_ARG Returned if an invalid hash type is given (see digest param), if prk is NULL, or if ikm is NULL and ikmLen is not 0
- MEMORY_E Returned if there is an error allocating memory
- HMAC_MIN_KEYLEN_E May be returned when using a FIPS implementation and the key length specified is shorter than the minimum acceptable FIPS standard
- WC_PENDING_E May be returned in a WOLF_CRYPTO_CB build when the registered crypto callback device has taken the request but not yet finished it; the caller re-invokes with identical

arguments until the result is no longer WC_PENDING_E. The TLS 1.3 key schedule does this in WOLFSSL_ASYNC_CRYPT builds.

Example

```
byte ikm[] = { // initialize with input keying material };
byte salt[] = { // initialize with optional salt };
byte prk[MAX_DIGEST_SIZE];

int ret = wc_Tls13_HKDF_Extract_ex(prk, salt, sizeof(salt), ikm,
    sizeof(ikm), WC_SHA512, NULL, INVALID_DEVID);
if ( ret != 0 ) {
    // error generating derived key
}
```

```
int wc_Tls13_HKDF_Expand_Label_ex(
    byte * okm,
    word32 okmLen,
    const byte * prk,
    word32 prkLen,
    const byte * protocol,
    word32 protocolLen,
    const byte * label,
    word32 labelLen,
    const byte * info,
    word32 infoLen,
    int digest,
    void * heap,
    int devId
)
```

Expand data using HMAC, salt and label and info. TLS v1.3 defines this function for key derivation. This is the _ex version adding heap hint and device identifier.

Parameters:

- **okm** Generated pseudorandom key - output key material.
- **okmLen** Length of generated pseudorandom key - output key material.
- **prk** Salt - pseudo-random key.
- **prkLen** Length of the salt - pseudo-random key.
- **protocol** TLS protocol label.
- **protocolLen** Length of the TLS protocol label.
- **info** Information to expand.
- **infoLen** Length of the information.
- **digest** Hash type to use for the HKDF. Valid types are: WC_SHA256, WC_SHA384 or WC_SHA512
- **heap** Heap hint to use for memory. Can be NULL
- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID (-2) if not used

See:

- wc_HKDF

- wc_HKDF_Extract
- wc_HKDF_Extract_ex
- wc_HKDF_Expand
- wc_Tls13_HKDF_Expand_Label
- wc_Tls13_HKDF_Expand_Label_Alloc

Return:

- 0 Returned upon successfully generating a key with the given inputs
- BAD_FUNC_ARG Returned if an invalid hash type is given (see type param)
- MEMORY_E Returned if there is an error allocating memory
- HMAC_MIN_KEYLEN_E May be returned when using a FIPS implementation and the key length specified is shorter than the minimum acceptable FIPS standard
- WC_PENDING_E May be returned in a WOLF_CRYPTO_CB build when the registered crypto callback device has taken the request but not yet finished it; the caller re-invokes with identical arguments until the result is no longer WC_PENDING_E. The TLS 1.3 key schedule does this in WOLFSSL_ASYNC_CRYPT builds.

```
int wc_Tls13_HKDF_Expand_Label(
    byte * okm,
    word32 okmLen,
    const byte * prk,
    word32 prkLen,
    const byte * protocol,
    word32 protocolLen,
    const byte * label,
    word32 labelLen,
    const byte * info,
    word32 infoLen,
    int digest
)
```

Expand data using HMAC, salt and label and info. TLS v1.3 defines this function for key derivation.

Parameters:

- **okm** Generated pseudorandom key - output key material.
- **okmLen** Length of generated pseudorandom key - output key material.
- **prk** Salt - pseudo-random key.
- **prkLen** Length of the salt - pseudo-random key.
- **protocol** TLS protocol label.
- **protocolLen** Length of the TLS protocol label.
- **info** Information to expand.
- **infoLen** Length of the information.
- **digest** Hash type to use for the HKDF. Valid types are: WC_SHA256, WC_SHA384 or WC_SHA512

See:

- wc_HKDF
- wc_HKDF_Extract
- wc_HKDF_Extract_ex

- wc_HKDF_Expand
- wc_Tls13_HKDF_Expand_Label_ex
- wc_Tls13_HKDF_Expand_Label_Alloc

Return:

- 0 Returned upon successfully generating a key with the given inputs
- BAD_FUNC_ARG Returned if an invalid hash type is given (see type param)
- MEMORY_E Returned if there is an error allocating memory
- HMAC_MIN_KEYLEN_E May be returned when using a FIPS implementation and the key length specified is shorter than the minimum acceptable FIPS standard

```
int wc_Tls13_HKDF_Expand_Label_Alloc(
    byte * okm,
    word32 okmLen,
    const byte * prk,
    word32 prkLen,
    const byte * protocol,
    word32 protocolLen,
    const byte * label,
    word32 labelLen,
    const byte * info,
    word32 infoLen,
    int digest,
    void * heap
)
```

This functions is very similar to wc_Tls13_HKDF_Expand_Label(), but it allocates memory if the stack space usually used isn't enough. Expand data using HMAC, salt and label and info. TLS v1.3 defines this function for key derivation. This is the _ex version adding heap hint and device identifier.

Parameters:

- **okm** Generated pseudorandom key - output key material.
- **okmLen** Length of generated pseudorandom key - output key material.
- **prk** Salt - pseudo-random key.
- **prkLen** Length of the salt - pseudo-random key.
- **protocol** TLS protocol label.
- **protocolLen** Length of the TLS protocol label.
- **info** Information to expand.
- **infoLen** Length of the information.
- **digest** Hash type to use for the HKDF. Valid types are: WC_SHA256, WC_SHA384 or WC_SHA512
- **heap** Heap hint to use for memory. Can be NULL

See:

- wc_HKDF
- wc_HKDF_Extract
- wc_HKDF_Extract_ex
- wc_HKDF_Expand
- wc_Tls13_HKDF_Expand_Label

- wc_Tls13_HKDF_Expand_Label_ex

Return:

- 0 Returned upon successfully generating a key with the given inputs
- BAD_FUNC_ARG Returned if an invalid hash type is given (see type param)
- MEMORY_E Returned if there is an error allocating memory
- HMAC_MIN_KEYLEN_E May be returned when using a FIPS implementation and the key length specified is shorter than the minimum acceptable FIPS standard
- WC_PENDING_E May be returned in a WOLF_CRYPTO_CB build when the registered crypto callback device has taken the request but not yet finished it; the caller re-invokes with identical arguments until the result is no longer WC_PENDING_E. The TLS 1.3 key schedule does this in WOLFSSL_ASYNC_CRYPT builds.

B.35 Algorithms - LMS / HSS (RFC 8554)

B.34.2.14 function wc_Tls13_HKDF_Expand_Label_Alloc More...

B.35.1 Functions

	Name
int	**wc_LmsKey_Init**(LmsKey * key, void * heap, int devId)Initializes an LmsKey object. Must be called before any other LMS/HSS operation. Use wc_LmsKey_Free() to release resources when done.
int	**wc_LmsKey_InitId**(LmsKey * key, const unsigned char * id, int len, void * heap, int devId)Initializes an LmsKey with a device_side key identifier. Equivalent to wc_LmsKey_Init() but also stashes a binary id that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.
int	**wc_LmsKey_InitLabel**(LmsKey * key, const char * label, void * heap, int devId)Initializes an LmsKey with a device_side key label. Equivalent to wc_LmsKey_Init() but also stashes a label string that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.
int	**wc_LmsKey_SetLmsParm**(LmsKey * key, enum wc_LmsParm lmsParm)Selects a predefined LMS/HSS parameter set by name. The enum wc_LmsParm encodes the tree depth (levels), per-tree height, Winternitz parameter and hash family in a single value. See the wc_LmsParm definition for the list of names available in a given build.

	Name
int	**wc_LmsKey_SetParameters**(LmsKey * key, int levels, int height, int winternitz)Sets the LMS/HSS parameters individually. The default SHA_256/256 hash is used. For finer control over the hash family use wc_LmsKey_SetParameters_ex().
int	**wc_LmsKey_SetParameters_ex**(LmsKey * key, int levels, int height, int winternitz, int hash)Sets the LMS/HSS parameters individually with an explicit hash family selector.
int	**wc_LmsKey_GetParameters**(const LmsKey * key, int * levels, int * height, int * winternitz)Retrieves the LMS/HSS parameters previously set on this key.
int	**wc_LmsKey_GetParameters_ex**(const LmsKey * key, int * levels, int * height, int * winternitz, int * hash)Retrieves the LMS/HSS parameters and hash family selector from this key.
int	**wc_LmsKey_SetWriteCb**(LmsKey * key, wc_lms_write_private_key_cb write_cb)Registers the callback that wolfSSL invokes to persist updated private key state. Because LMS/HSS is stateful, the application MUST persist the private key after each successful sign before the signature is released to a peer; otherwise a crash or restart can lead to one-time key reuse and break the scheme.
int	**wc_LmsKey_SetReadCb**(LmsKey * key, wc_lms_read_private_key_cb read_cb)Registers the callback that wolfSSL invokes to load persisted private key state. Used by wc_LmsKey_Reload() to bring a saved key back into memory for further signing.
int	**wc_LmsKey_SetContext**(LmsKey * key, void * context)Sets the opaque context pointer passed to both the read and write private-key callbacks. Typically used to carry a file handle, database connection, or other persistence-layer state.
int	**wc_LmsKey_MakeKey**(LmsKey * key, WC_RNG * rng)Generates a fresh LMS/HSS key pair. Parameters must already be set (via wc_LmsKey_SetLmsParm() or wc_LmsKey_SetParameters()) and read/write callbacks must be registered. The newly generated private key is persisted via the write callback before the function returns; on success the key transitions to state WC_LMS_STATE_OK.

	Name
int	**wc_LmsKey_Reload**(LmsKey * key)Reloads a previously generated LMS/HSS private key from persistent storage using the registered read callback, restoring the key to a state where it can sign further messages. On success the key is in state WC_LMS_STATE_OK.
int	**wc_LmsKey_GetPrivLen**(const LmsKey * key, word32 * len)Returns the size in bytes of the encoded private key for the parameter set selected on this key.
int	**wc_LmsKey_GetPubLen**(const LmsKey * key, word32 * len)Returns the size in bytes of the LMS/HSS public key for the parameter set selected on this key.
int	**wc_LmsKey_GetSigLen**(const LmsKey * key, word32 * len)Returns the LMS/HSS signature size in bytes for the parameter set selected on this key.
int	**wc_LmsKey_Sign**(LmsKey * key, byte * sig, word32 * sigSz, const byte * msg, int msgSz)Signs msg with the LMS/HSS private key in key. On entry sigSz is the size of the sig buffer; on success it is updated to the bytes written.
int	**wc_LmsKey_SigsLeft**(LmsKey * key)Reports whether this key can create another signature. It does not return a count. A key whose one-time keys are used up returns 0 and should be retired.
void	**wc_LmsKey_Free**(LmsKey * key)Releases resources held by an LmsKey. Safe to call with a NULL pointer. After this call the key is in state WC_LMS_STATE_FREED and must be re-initialized before reuse.
int	**wc_LmsKey_ExportPub**(LmsKey * keyDst, const LmsKey * keySrc)Copies the public part of keySrc into keyDst. The destination key inherits the same parameters and may be used for verification; it does not carry the private key state and cannot sign. Useful for handing a verifier the minimal data it needs.
int	**wc_LmsKey_ExportPub_ex**(LmsKey * keyDst, const LmsKey * keySrc, void * heap, int devId)Like wc_LmsKey_ExportPub() but the destination key is initialized fresh with the supplied heap and devId.
int	**wc_LmsKey_ExportPubRaw**(const LmsKey * key, byte * out, word32 * outLen)Exports the LMS/HSS public key as a raw byte string. On entry *outLen is the size of out; on success it is updated to the bytes written.

	Name
int	**wc_LmsKey_ImportPubRaw**(LmsKey * key, const byte * in, word32 inLen)Imports a raw LMS/HSS public key into key. The key must be in state WC_LMS_STATE_INITED. Parameter information is recovered from the encoded header, after which the key transitions to state WC_LMS_STATE_VERIFYONLY.
int	**wc_LmsKey_Verify**(LmsKey * key, const byte * sig, word32 sigSz, const byte * msg, int msgSz)Verifies an LMS/HSS signature against msg using the public key held in key. The function returns 0 only when the signature is valid; any other value indicates the signature was rejected.
const char *	**wc_LmsKey_ParmToStr**(enum wc_LmsParm lmsParm)Returns a static, NUL-terminated string describing an LMS parameter set. Useful for logging and diagnostics.
int	**wc_LmsKey_GetKid**(LmsKey * key, const byte ** kid, word32 * kidSz)Returns a pointer to the 16_byte LMS Key Identifier (I) embedded in the private key, together with its length. The returned pointer aliases internal key memory and is valid only until the key is freed.
const byte *	**wc_LmsKey_GetKidFromPrivRaw**(const byte * priv, word32 privSz)Returns a pointer to the LMS Key Identifier (I) within a raw encoded private key buffer, without requiring an LmsKey object. Used to look up the matching state record in persistent storage during reload.

B.35.2 Detailed Description

LMS (Leighton-Micali Signatures) and its multi-tree composition HSS (Hierarchical Signature System) are stateful hash-based signature schemes specified in RFC 8554 and NIST SP 800-208. Each signature consumes a one-time component of the private key, so the application MUST persist the private key state (via the read/write callbacks registered with wc_LmsKey_SetReadCb() and wc_LmsKey_SetWriteCb()) between signing operations. Reusing a one-time key destroys the security of the scheme.

The number of signatures available from a key is bounded by the parameter set; check whether any remain with wc_LmsKey_SigsLeft().

B.35.3 Functions Documentation

```
int wc_LmsKey_Init(
    LmsKey * key,
    void * heap,
    int devId
)
```

Initializes an LmsKey object. Must be called before any other LMS/HSS operation. Use wc_LmsKey_Free() to release resources when done.

Parameters:

- **key** Pointer to the LmsKey to initialize.
- **heap** Heap hint for dynamic memory allocation. May be NULL.
- **devId** Device identifier for hardware crypto callbacks. Use INVALID_DEVID for software-only.

See:

- wc_LmsKey_Free
- wc_LmsKey_SetLmsParm
- wc_LmsKey_SetParameters
- wc_LmsKey_MakeKey

Return:

- 0 on success.
- BAD_FUNC_ARG if key is NULL.

LMS (Leighton-Micali Signatures) and the HSS multi-tree composition (RFC 8554, NIST SP 800-208) are STATEFUL hash-based signature schemes: each call to wc_LmsKey_Sign() consumes a one_time component of the private key, and reusing a one_time key breaks the security of the scheme. Applications MUST persist the private key state between sign calls; see wc_LmsKey_SetWriteCb() and wc_LmsKey_SetReadCb().

After init the key is in state WC_LMS_STATE_INITED. The parameters must be set with wc_LmsKey_SetLmsParm() or wc_LmsKey_SetParameters() before generating or reloading a key.

Example

```
LmsKey key;
int ret;

ret = wc_LmsKey_Init(&key, NULL, INVALID_DEVID);
if (ret != 0) {
    // error initializing key
}
wc_LmsKey_SetLmsParm(&key, WC_LMS_PARM_L2_H10_W8);
// ... use key ...
wc_LmsKey_Free(&key);
```

```
int wc_LmsKey_InitId(
    LmsKey * key,
    const unsigned char * id,
    int len,
    void * heap,
    int devId
)
```

Initializes an LmsKey with a device-side key identifier. Equivalent to wc_LmsKey_Init() but also stashes a binary id that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.

Parameters:

- **key** Pointer to the LmsKey to initialize.
- **id** Pointer to the device-side key identifier bytes. May be NULL when len is 0.
- **len** Number of bytes in id; must be in [0, LMS_MAX_ID_LEN].
- **heap** Heap hint for dynamic memory allocation.
- **devId** Device identifier for the crypto callback.

See:

- wc_LmsKey_Init
- wc_LmsKey_InitLabel
- wc_LmsKey_Free

Return:

- 0 on success.
- BAD_FUNC_ARG if key is NULL, or id is NULL while len > 0.
- BUFFER_E if len is negative or greater than LMS_MAX_ID_LEN.

The id is copied into the key object; the caller may free its buffer immediately after this call returns.

```
int wc_LmsKey_InitLabel(
    LmsKey * key,
    const char * label,
    void * heap,
    int devId
)
```

Initializes an LmsKey with a device-side key label. Equivalent to wc_LmsKey_Init() but also stashes a label string that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.

Parameters:

- **key** Pointer to the LmsKey to initialize.
- **label** NUL-terminated device-side key label.
- **heap** Heap hint for dynamic memory allocation.
- **devId** Device identifier for the crypto callback.

See:

- wc_LmsKey_Init
- wc_LmsKey_InitId

Return:

- 0 on success.
- BAD_FUNC_ARG if key or label is NULL.

- BUFFER_E if label is empty or longer than LMS_MAX_LABEL_LEN.

```
int wc_LmsKey_SetLmsParm(
    LmsKey * key,
    enum wc_LmsParm lmsParm
)
```

Selects a predefined LMS/HSS parameter set by name. The enum wc_LmsParm encodes the tree depth (levels), per-tree height, Winternitz parameter and hash family in a single value. See the wc_LmsParm definition for the list of names available in a given build.

Parameters:

- **key** Pointer to an LmsKey in state WC_LMS_STATE_INITED.
- **lmsParm** A wc_LmsParm constant (e.g. WC_LMS_PARM_L2_H10_W8).

See:

- wc_LmsKey_SetParameters
- wc_LmsKey_GetParameters
- wc_LmsKey_ParmToStr

Return:

- 0 on success.
- BAD_FUNC_ARG if key is NULL, or if lmsParm is not recognized or names a parameter set that was not compiled in.
- BAD_STATE_E if key is not in state WC_LMS_STATE_INITED.

Example

```
LmsKey key;

wc_LmsKey_Init(&key, NULL, INVALID_DEVID);
wc_LmsKey_SetLmsParm(&key, WC_LMS_PARM_L2_H10_W8);
```

```
int wc_LmsKey_SetParameters(
    LmsKey * key,
    int levels,
    int height,
    int winternitz
)
```

Sets the LMS/HSS parameters individually. The default SHA-256/256 hash is used. For finer control over the hash family use wc_LmsKey_SetParameters_ex().

Parameters:

- **key** Pointer to an LmsKey in state WC_LMS_STATE_INITED.
- **levels** Number of Merkle-tree levels in the HSS chain.
- **height** Height of each individual Merkle tree.
- **winternitz** Winternitz parameter (1, 2, 4 or 8).

See:

- wc_LmsKey_SetParameters_ex
- wc_LmsKey_SetLmsParm
- wc_LmsKey_GetParameters

Return:

- 0 on success.
- BAD_FUNC_ARG if key is NULL, or if the requested parameter combination is not supported in this build.
- BAD_STATE_E if key is not in state WC_LMS_STATE_INITED.

The combination of parameters must match one of the allowed sets in RFC 8554:

- levels: 1..8
- height: 5, 10, 15, 20 (and 25 in some builds)
- winternitz: 1, 2, 4, or 8

The maximum number of signatures available from a key is 2^(levels * height).

```
int wc_LmsKey_SetParameters_ex(
    LmsKey * key,
    int levels,
    int height,
    int winternitz,
    int hash
)
```

Sets the LMS/HSS parameters individually with an explicit hash family selector.

Parameters:

- **key** Pointer to an LmsKey in state WC_LMS_STATE_INITED.
- **levels** Number of Merkle-tree levels.
- **height** Height of each tree.
- **winternitz** Winternitz parameter (1, 2, 4 or 8).
- **hash** Hash family selector identifying SHA-256/256, SHA-256/192, SHAKE256/256 or SHAKE256/192, as supported by the build.

See:

- wc_LmsKey_SetParameters
- wc_LmsKey_GetParameters_ex

Return:

- 0 on success.
- BAD_FUNC_ARG if key is NULL, or if the requested parameter combination is not supported in this build.
- BAD_STATE_E if key is not in state WC_LMS_STATE_INITED.

```
int wc_LmsKey_GetParameters(
    const LmsKey * key,
    int * levels,
    int * height,
    int * winternitz
)
```

Retrieves the LMS/HSS parameters previously set on this key.

Parameters:

- **key** Pointer to an LmsKey with parameters set.
- **levels** Receives the number of tree levels.
- **height** Receives the per-tree height.
- **winternitz** Receives the Winternitz parameter.

See:

- wc_LmsKey_SetParameters
- wc_LmsKey_GetParameters_ex

Return:

- 0 on success.
- BAD_FUNC_ARG if any pointer is NULL or no parameters are set.

```
int wc_LmsKey_GetParameters_ex(
    const LmsKey * key,
    int * levels,
    int * height,
    int * winternitz,
    int * hash
)
```

Retrieves the LMS/HSS parameters and hash family selector from this key.

Parameters:

- **key** Pointer to an LmsKey with parameters set.
- **levels** Receives the number of tree levels.

- **height** Receives the per-tree height.
- **winternitz** Receives the Winternitz parameter.
- **hash** Receives the hash family selector.

See: wc_LmsKey_SetParameters_ex

Return:

- 0 on success.
- BAD_FUNC_ARG if any pointer is NULL or no parameters are set.

```
int wc_LmsKey_SetWriteCb(
    LmsKey * key,
    wc_lms_write_private_key_cb write_cb
)
```

Registers the callback that wolfSSL invokes to persist updated private key state. Because LMS/HSS is stateful, the application MUST persist the private key after each successful sign before the signature is released to a peer; otherwise a crash or restart can lead to one-time key reuse and break the scheme.

Parameters:

- **key** Pointer to an LmsKey.
- **write_cb** Callback invoked to persist the private key.

See:

- wc_LmsKey_SetReadCb
- wc_LmsKey_SetContext
- wc_LmsKey_Sign

Return:

- 0 on success.
- BAD_FUNC_ARG if key or write_cb is NULL.

The callback receives the encoded private key bytes and returns one of the wc_LmsRc codes. WC_LMS_RC_SAVED_TO_NV_MEMORY signals a durable write; other return codes are treated as failures.

```
int wc_LmsKey_SetReadCb(
    LmsKey * key,
    wc_lms_read_private_key_cb read_cb
)
```

Registers the callback that wolfSSL invokes to load persisted private key state. Used by wc_LmsKey_Reload() to bring a saved key back into memory for further signing.

Parameters:

- **key** Pointer to an LmsKey.
- **read_cb** Callback invoked to load the private key.

See:

- wc_LmsKey_SetWriteCb
- wc_LmsKey_SetContext
- wc_LmsKey_Reload

Return:

- 0 on success.
- BAD_FUNC_ARG if key or read_cb is NULL.

```
int wc_LmsKey_SetContext(
    LmsKey * key,
    void * context
)
```

Sets the opaque context pointer passed to both the read and write private-key callbacks. Typically used to carry a file handle, database connection, or other persistence-layer state.

Parameters:

- **key** Pointer to an LmsKey.
- **context** Application-defined pointer; may be NULL.

See:

- wc_LmsKey_SetReadCb
- wc_LmsKey_SetWriteCb

Return:

- 0 on success.
- BAD_FUNC_ARG if key is NULL.

```
int wc_LmsKey_MakeKey(
    LmsKey * key,
    WC_RNG * rng
)
```

Generates a fresh LMS/HSS key pair. Parameters must already be set (via wc_LmsKey_SetLmsParm() or wc_LmsKey_SetParameters()) and read/write callbacks must be registered. The newly generated private key is persisted via the write callback before the function returns; on success the key transitions to state WC_LMS_STATE_OK.

Parameters:

- **key** Pointer to an LmsKey in state WC_LMS_STATE_PARMSET with callbacks set.
- **rng** Pointer to an initialized WC_RNG.

See:

- wc_LmsKey_Sign
- wc_LmsKey_Reload

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- MEMORY_E on allocation failure.

Key generation runtime grows quickly with the first tree's height: a 3-level h=5 tree is much faster than a 1-level h=15 tree even though both yield the same total signature count.

Example

```
LmsKey key;
WC_RNG rng;

wc_LmsKey_Init(&key, NULL, INVALID_DEVID);
wc_LmsKey_SetLmsParm(&key, WC_LMS_PARM_L2_H10_W8);
wc_LmsKey_SetWriteCb(&key, my_write_cb);
wc_LmsKey_SetReadCb(&key, my_read_cb);
wc_LmsKey_SetContext(&key, &my_storage);
wc_InitRng(&rng);

if (wc_LmsKey_MakeKey(&key, &rng) != 0) {
    // error generating key
}
```

```
int wc_LmsKey_Reload(
    LmsKey * key
)
```

Reloads a previously generated LMS/HSS private key from persistent storage using the registered read callback, restoring the key to a state where it can sign further messages. On success the key is in state WC_LMS_STATE_OK.

Parameters:

- **key** Pointer to an LmsKey with parameters and read callback set.

See:

- wc_LmsKey_MakeKey
- wc_LmsKey_SetReadCb

Return:

- 0 on success, including the device-backed no-op.
- BAD_FUNC_ARG if any required pointer is NULL.
- WC_LMS_RC_* mapped error if the read callback fails.
- IO_FAILED_E if the private key could not be read.
- NO_VALID_DEVID with WOLF_CRYPTO_CB_ONLY_LMS when the key is not device-backed.

The same parameters set at key-generation time must be reapplied to the LmsKey before calling Reload (the persisted blob is just the private key bytes; the parameter set is metadata the application owns).

With WOLF_CRYPTO_CB, a key on a devId with no read callback is taken to be device-backed and Reload is a no-op. With a read callback the software reload runs as usual.

WOLF_CRYPTO_CB_ONLY_LMS removes the software reload, so the read callback is never called and only the device-backed no-op succeeds.

A reloaded key holds no public key: neither arm populates it. Call wc_LmsKey_ImportPubRaw() on a separate key to verify, or keep the public key exported at generation time.

```
int wc_LmsKey_GetPrivLen(
    const LmsKey * key,
    word32 * len
)
```

Returns the size in bytes of the encoded private key for the parameter set selected on this key.

Parameters:

- **key** Pointer to an LmsKey with parameters set.
- **len** Receives the private key size in bytes.

See:

- wc_LmsKey_GetPubLen
- wc_LmsKey_GetSigLen

Return:

- 0 on success.
- BAD_FUNC_ARG if key or len is NULL.

```
int wc_LmsKey_GetPubLen(
    const LmsKey * key,
    word32 * len
)
```

Returns the size in bytes of the LMS/HSS public key for the parameter set selected on this key.

Parameters:

- **key** Pointer to an LmsKey with parameters set.
- **len** Receives the public key size in bytes.

See:

- wc_LmsKey_ExportPubRaw
- wc_LmsKey_GetPrivLen

Return:

- 0 on success.
- BAD_FUNC_ARG if key or len is NULL.

```
int wc_LmsKey_GetSigLen(
    const LmsKey * key,
    word32 * len
)
```

Returns the LMS/HSS signature size in bytes for the parameter set selected on this key.

Parameters:

- **key** Pointer to an LmsKey with parameters set.
- **len** Receives the signature size in bytes.

See: wc_LmsKey_Sign

Return:

- 0 on success.
- BAD_FUNC_ARG if key or len is NULL.

```
int wc_LmsKey_Sign(
    LmsKey * key,
    byte * sig,
    word32 * sigSz,
    const byte * msg,
    int msgSz
)
```

Signs msg with the LMS/HSS private key in key. On entry sigSz is the size of the sig buffer; on success it is updated to the bytes written.

Parameters:

- **key** Pointer to an LmsKey in state WC_LMS_STATE_OK.
- **sig** Buffer that receives the signature.
- **sigSz** In: size of sig. Out: bytes written.
- **msg** Message to sign.
- **msgSz** Length of msg in bytes.

See:

- wc_LmsKey_Verify
- wc_LmsKey_SigsLeft
- wc_LmsKey_SetWriteCb

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- BUFFER_E if *sigSz is smaller than the signature size.
- -1 (or similar) if all one-time keys have been used.

Each successful sign call consumes a one-time component of the private key. The updated key state is persisted via the registered write callback BEFORE the new signature is returned to the caller. If the write callback fails the sign call fails and the signature is not released. When the supply of one-time keys is exhausted the key transitions to state WC_LMS_STATE_NOSIGS and further sign attempts return SIG_OTHER_E (or similar) – query wc_LmsKey_SigsLeft() to detect this condition in advance.

```
int wc_LmsKey_SigsLeft(
    LmsKey * key
)
```

Reports whether this key can create another signature. It does not return a count. A key whose one-time keys are used up returns 0 and should be retired.

Parameters:

- **key** Pointer to an LmsKey in state WC_LMS_STATE_OK.

See: wc_LmsKey_Sign

Return:

- 1 if at least one signature remains.
- 0 if the one-time keys are exhausted, if key is NULL, has no parameters set or is verify-only, or if a crypto callback device returns an error.

```
void wc_LmsKey_Free(
    LmsKey * key
)
```

Releases resources held by an LmsKey. Safe to call with a NULL pointer. After this call the key is in state WC_LMS_STATE_FREED and must be re-initialized before reuse.

Parameters:

- **key** Pointer to the LmsKey to free.

See: wc_LmsKey_Init

```
int wc_LmsKey_ExportPub(
    LmsKey * keyDst,
    const LmsKey * keySrc
)
```

Copies the public part of keySrc into keyDst. The destination key inherits the same parameters and may be used for verification; it does not carry the private key state and cannot sign. Useful for handing a verifier the minimal data it needs.

Parameters:

- **keyDst** Pointer to an initialized destination LmsKey.
- **keySrc** Pointer to an LmsKey with the public key.

See:

- wc_LmsKey_ExportPub_ex
- wc_LmsKey_ExportPubRaw

Return:

- 0 on success.
- BAD_FUNC_ARG if keyDst or keySrc is NULL.
- BAD_STATE_E if keySrc holds no public key.

```
int wc_LmsKey_ExportPub_ex(
    LmsKey * keyDst,
    const LmsKey * keySrc,
    void * heap,
    int devId
)
```

Like wc_LmsKey_ExportPub() but the destination key is initialized fresh with the supplied heap and devId.

Parameters:

- **keyDst** Pointer to an LmsKey to populate.
- **keySrc** Pointer to an LmsKey with the public key.

- **heap** Heap hint for keyDst.
- **devId** Device identifier for keyDst.

See: wc_LmsKey_ExportPub

Return:

- 0 on success.
- BAD_FUNC_ARG if keyDst or keySrc is NULL.
- BAD_STATE_E if keySrc holds no public key.

```
int wc_LmsKey_ExportPubRaw(
    const LmsKey * key,
    byte * out,
    word32 * outLen
)
```

Exports the LMS/HSS public key as a raw byte string. On entry *outLen is the size of out; on success it is updated to the bytes written.

Parameters:

- **key** Pointer to an LmsKey.
- **out** Buffer that receives the public key.
- **outLen** In: size of out. Out: bytes written.

See:

- wc_LmsKey_ImportPubRaw
- wc_LmsKey_GetPubLen

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- BAD_STATE_E if key holds no public key.
- BUFFER_E if *outLen is smaller than the public key size.

```
int wc_LmsKey_ImportPubRaw(
    LmsKey * key,
    const byte * in,
    word32 inLen
)
```

Imports a raw LMS/HSS public key into key. The key must be in state WC_LMS_STATE_INITED. Parameter information is recovered from the encoded header, after which the key transitions to state WC_LMS_STATE_VERIFYONLY.

Parameters:

- **key** Pointer to an LmsKey in WC_LMS_STATE_INITED.
- **in** Raw public key bytes.
- **inLen** Length of in in bytes.

See:

- wc_LmsKey_ExportPubRaw
- wc_LmsKey_Verify

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- BUFFER_E if inLen is too small.

```
int wc_LmsKey_Verify(
    LmsKey * key,
    const byte * sig,
    word32 sigSz,
    const byte * msg,
    int msgSz
)
```

Verifies an LMS/HSS signature against msg using the public key held in key. The function returns 0 only when the signature is valid; any other value indicates the signature was rejected.

Parameters:

- **key** Pointer to an LmsKey with the public key set.
- **sig** Signature bytes to verify.
- **sigSz** Length of sig in bytes.
- **msg** Message that was signed.
- **msgSz** Length of msg in bytes.

See:

- wc_LmsKey_Sign
- wc_LmsKey_ImportPubRaw

Return:

- 0 on a valid signature.
- BAD_FUNC_ARG if any required pointer is NULL.
- BAD_STATE_E if key holds no public key.
- SIG_VERIFY_E (or similar) if the signature is invalid or malformed.

```
const char * wc_LmsKey_ParmToStr(
    enum wc_LmsParm lmsParm
)
```

Returns a static, NUL-terminated string describing an LMS parameter set. Useful for logging and diagnostics.

Parameters:

- **lmsParm** A wc_LmsParm constant.

See: wc_LmsKey_SetLmsParm

Return:

- Pointer to a static string on success.
- NULL if lmsParm is not recognized.

```
int wc_LmsKey_GetKid(
    LmsKey * key,
    const byte ** kid,
    word32 * kidSz
)
```

Returns a pointer to the 16-byte LMS Key Identifier (I) embedded in the private key, together with its length. The returned pointer aliases internal key memory and is valid only until the key is freed.

Parameters:

- **key** Pointer to an LmsKey with a private key.
- **kid** Receives a pointer to the I bytes.
- **kidSz** Receives the length (16 / WC_LMS_I_LEN).

See: wc_LmsKey_GetKidFromPrivRaw

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- BAD_STATE_E if the private key is not in memory, such as when a device keeps it.

```
const byte * wc_LmsKey_GetKidFromPrivRaw(
    const byte * priv,
    word32 privSz
)
```

Returns a pointer to the LMS Key Identifier (I) within a raw encoded private key buffer, without requiring an LmsKey object. Used to look up the matching state record in persistent storage during reload.

Parameters:

- **priv** Encoded private key bytes.
- **privSz** Length of priv in bytes.

See: wc_LmsKey_GetKid

Return:

- Pointer to the I bytes within priv on success.
- NULL if priv is NULL or privSz is too small to contain a valid header.

B.36 Algorithms - MD2

B.35.3.27 function wc_LmsKey_GetKidFromPrivRaw

B.36.1 Functions

	Name
int	**wc_InitMd2**(wc_Md2 * md2)This function initializes md2. This is automatically called by wc_Md2Hash.
int	**wc_Md2Update**(wc_Md2 * md2, const byte * data, word32 len)Can be called to continually hash the provided byte array of length len.
int	**wc_Md2Final**(wc_Md2 * md2, byte * hash)Finalizes hashing of data. Result is placed into hash.
int	**wc_Md2Hash**(const byte * data, word32 len, byte * hash)Convenience function, handles all the hashing and places the result into hash.

B.36.2 Functions Documentation

```
int wc_InitMd2(
    wc_Md2 * md2
)
```

This function initializes md2. This is automatically called by wc_Md2Hash.

Parameters:

- **md2** pointer to the md2 structure to use for encryption

See:

- wc_Md2Hash
- wc_Md2Update
- wc_Md2Final

Return:

- 0 Returned upon successfully initializing
- BAD_FUNC_ARG Returned if md2 is NULL

Example

```
md2 md2[1];
if ((ret = wc_InitMd2(md2)) != 0) {
   WOLFSSL_MSG("wc_Initmd2 failed");
}
else {
   wc_Md2Update(md2, data, len);
   wc_Md2Final(md2, hash);
}
```

```
int wc_Md2Update(
    wc_Md2 * md2,
    const byte * data,
    word32 len
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **md2** pointer to the md2 structure to use for encryption
- **data** the data to be hashed
- **len** length of data to be hashed

See:

- wc_Md2Hash
- wc_Md2Final
- wc_InitMd2

Return:

- 0 Returned upon successfully adding the data to the digest.
- BAD_FUNC_ARG Returned if md2 is NULL, or if data is NULL and len is non-zero

Example

```
md2 md2[1];
byte data[] = { }; // Data to be hashed
word32 len = sizeof(data);

if ((ret = wc_InitMd2(md2)) != 0) {
   WOLFSSL_MSG("wc_Initmd2 failed");
```

```
}
else {
   wc_Md2Update(md2, data, len);
   wc_Md2Final(md2, hash);
}
```

```
int wc_Md2Final(
    wc_Md2 * md2,
    byte * hash
)
```

Finalizes hashing of data. Result is placed into hash.

Parameters:

- **md2** pointer to the md2 structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Md2Hash
- wc_Md2Final
- wc_InitMd2

Return:

- 0 Returned upon successfully finalizing.
- BAD_FUNC_ARG Returned if md2 or hash is NULL

Example

```
md2 md2[1];
byte data[] = { }; // Data to be hashed
word32 len = sizeof(data);

if ((ret = wc_InitMd2(md2)) != 0) {
   WOLFSSL_MSG("wc_Initmd2 failed");
}
else {
   wc_Md2Update(md2, data, len);
   wc_Md2Final(md2, hash);
}
```

```
int wc_Md2Hash(
    const byte * data,
    word32 len,
    byte * hash
)
```

Convenience function, handles all the hashing and places the result into hash.

Parameters:

- **data** the data to hash
- **len** the length of data
- **hash** Byte array to hold hash value.

See:

- wc_Md2Hash
- wc_Md2Final
- wc_InitMd2

Return:

- 0 Returned upon successfully hashing the data.
- Memory_E memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Example

none

B.37 Algorithms - MD4

B.36.2.4 function wc_Md2Hash

B.37.1 Functions

	Name
int	**wc_InitMd4**(wc_Md4 * md4)This function initializes md4. This is automatically called by wc_Md4Hash.
int	**wc_Md4Update**(wc_Md4 * md4, const byte * data, word32 len)Can be called to continually hash the provided byte array of length len.
int	**wc_Md4Final**(wc_Md4 * md4, byte * hash)Finalizes hashing of data. Result is placed into hash.

B.37.2 Functions Documentation

```
int wc_InitMd4(
    wc_Md4 * md4
)
```

This function initializes md4. This is automatically called by wc_Md4Hash.

Parameters:

- **md4** pointer to the md4 structure to use for encryption

See:

- wc_Md4Hash
- wc_Md4Update
- wc_Md4Final

Return:

- 0 Returned upon successfully initializing
- BAD_FUNC_ARG Returned if md4 is NULL

Example

```
md4 md4[1];
if ((ret = wc_InitMd4(md4)) != 0) {
   WOLFSSL_MSG("wc_Initmd4 failed");
}
else {
   wc_Md4Update(md4, data, len);
   wc_Md4Final(md4, hash);
}
```

```
int wc_Md4Update(
    wc_Md4 * md4,
    const byte * data,
    word32 len
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **md4** pointer to the md4 structure to use for encryption
- **data** the data to be hashed
- **len** length of data to be hashed

See:

- wc_Md4Hash
- wc_Md4Final
- wc_InitMd4

Return:

- 0 Returned upon successfully adding the data to the digest.
- BAD_FUNC_ARG Returned if md4 is NULL, or if data is NULL and len is non-zero

Example

```
md4 md4[1];
byte data[] = { }; // Data to be hashed
word32 len = sizeof(data);

if ((ret = wc_InitMd4(md4)) != 0) {
   WOLFSSL_MSG("wc_Initmd4 failed");
}
else {
   wc_Md4Update(md4, data, len);
   wc_Md4Final(md4, hash);
}
```

```
int wc_Md4Final(
    wc_Md4 * md4,
    byte * hash
)
```

Finalizes hashing of data. Result is placed into hash.

Parameters:

- **md4** pointer to the md4 structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Md4Hash
- wc_Md4Final
- wc_InitMd4

Return:

- 0 Returned upon successfully finalizing.
- BAD_FUNC_ARG Returned if md4 or hash is NULL

Example

```
md4 md4[1];
if ((ret = wc_InitMd4(md4)) != 0) {
    WOLFSSL_MSG("wc_Initmd4 failed");
}
else {
```

```
    wc_Md4Update(md4, data, len);
    wc_Md4Final(md4, hash);
}
```

B.38 Algorithms - MD5

B.37.2.3 function wc_Md4Final

B.38.1 Functions

	Name
int	**wc_Md5Hash**(const byte * data, word32 len, byte * hash)Convenience function, handles all the hashing and places the result into hash.
int	**wc_InitMd5**(wc_Md5 * md5)This function initializes md5. This is automatically called by wc_Md5Hash.
int	**wc_Md5Update**(wc_Md5 * md5, const byte * data, word32 len)Can be called to continually hash the provided byte array of length len.
int	**wc_Md5Final**(wc_Md5 * md5, byte * hash)Finalizes hashing of data. Result is placed into hash. Md5 Struct is reset. Note: This function will also return the result of calling IntelQaSymMd5() in the case that HAVE_INTEL_QA is defined.
void	**wc_Md5Free**(wc_Md5 * md5)Resets the Md5 structure. Note: this is only supported if you have WOLFSSL_TI_HASH defined.
int	**wc_Md5GetHash**(wc_Md5 * md5, byte * hash)Gets hash data. Result is placed into hash. Md5 struct is not reset.

B.38.2 Functions Documentation

```
int wc_Md5Hash(
    const byte * data,
    word32 len,
    byte * hash
)
```

Convenience function, handles all the hashing and places the result into hash.

Parameters:

- **data** the data to hash
- **len** the length of data
- **hash** Byte array to hold hash value.
- **hashLen** Number of bytes to write to hash.

See:

- wc_Md5Hash
- wc_Md5Final
- wc_InitMd5

Return:

- 0 Returned upon successfully hashing the data.
- Memory_E memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Example

```
const byte* data;
word32 data_len;
byte* hash;
int ret;
...
ret = wc_Md5Hash(data, data_len, hash);
if (ret != 0) {
     // Md5 Hash Failure Case.
}
```

```
int wc_InitMd5(
    wc_Md5 * md5
)
```

This function initializes md5. This is automatically called by wc_Md5Hash.

Parameters:

- **md5** pointer to the md5 structure to use for encryption

See:

- wc_Md5Hash
- wc_Md5Update
- wc_Md5Final

Return:

- 0 Returned upon successfully initializing.
- BAD_FUNC_ARG Returned if the Md5 structure is passed as a NULL value.

Example

```
Md5 md5;
byte* hash;
if ((ret = wc_InitMd5(&md5)) != 0) {
   WOLFSSL_MSG("wc_Initmd5 failed");
}
else {
   ret = wc_Md5Update(&md5, data, len);
   if (ret != 0) {
     // Md5 Update Failure Case.
   }
   ret = wc_Md5Final(&md5, hash);
  if (ret != 0) {
    // Md5 Final Failure Case.
  }
}
```

```
int wc_Md5Update(
    wc_Md5 * md5,
    const byte * data,
    word32 len
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **md5** pointer to the md5 structure to use for encryption
- **data** the data to be hashed
- **len** length of data to be hashed

See:

- wc_Md5Hash
- wc_Md5Final
- wc_InitMd5

Return:

- 0 Returned upon successfully adding the data to the digest.
- BAD_FUNC_ARG Returned if the Md5 structure is NULL or if data is NULL and len is greater than zero. The function should not return an error if the data parameter is NULL and len is zero.

Example

```
Md5 md5;
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitMd5(&md5)) != 0) {
   WOLFSSL_MSG("wc_Initmd5 failed");
```

```
}
else {
   ret = wc_Md5Update(&md5, data, len);
   if (ret != 0) {
     // Md5 Update Error Case.
   }
   ret = wc_Md5Final(&md5, hash);
   if (ret != 0) {
    // Md5 Final Error Case.
   }
}
```

```
int wc_Md5Final(
    wc_Md5 * md5,
    byte * hash
)
```

Finalizes hashing of data. Result is placed into hash. Md5 Struct is reset. Note: This function will also return the result of calling IntelQaSymMd5() in the case that HAVE_INTEL_QA is defined.

Parameters:

- **md5** pointer to the md5 structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Md5Hash
- wc_InitMd5
- wc_Md5GetHash

Return:

- 0 Returned upon successfully finalizing.
- BAD_FUNC_ARG Returned if the Md5 structure or hash pointer is passed in NULL.

Example

```
md5 md5[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitMd5(md5)) != 0) {
   WOLFSSL_MSG("wc_Initmd5 failed");
}
else {
   ret = wc_Md5Update(md5, data, len);
   if (ret != 0) {
    // Md5 Update Failure Case.
   }
```

```
  ret = wc_Md5Final(md5, hash);
   if (ret != 0) {
    // Md5 Final Failure Case.
   }
}
```

```
void wc_Md5Free(
    wc_Md5 * md5
)
```

Resets the Md5 structure. Note: this is only supported if you have WOLFSSL_TI_HASH defined.

Parameters:

- **md5** Pointer to the Md5 structure to be reset.

See:

- wc_InitMd5
- wc_Md5Update
- wc_Md5Final

Return: none No returns.

Example

```
Md5 md5;
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitMd5(&md5)) != 0) {
    WOLFSSL_MSG("wc_InitMd5 failed");
}
else {
    wc_Md5Update(&md5, data, len);
    wc_Md5Final(&md5, hash);
    wc_Md5Free(&md5);
}
```

```
int wc_Md5GetHash(
    wc_Md5 * md5,
    byte * hash
)
```

Gets hash data. Result is placed into hash. Md5 struct is not reset.

Parameters:

- **md5** pointer to the md5 structure to use for encryption.

- **hash** Byte array to hold hash value.

See:

- wc_Md5Hash
- wc_Md5Final
- wc_InitMd5

Return: none No returns

Example

```
md5 md5[1];
if ((ret = wc_InitMd5(md5)) != 0) {
   WOLFSSL_MSG("wc_Initmd5 failed");
}
else {
   wc_Md5Update(md5, data, len);
   wc_Md5GetHash(md5, hash);
}
```

B.39 Algorithms - ML-DSA (FIPS 204)

B.38.2.6 function wc_Md5GetHash More...

B.39.1 Functions

	Name
int	**wc_MlDsaKey_Init**(wc_MlDsaKey * key, void * heap, int devId)Initializes a wc_MlDsaKey object. Must be called before any other ML_DSA operation. Use wc_MlDsaKey_Free() to release resources when done.
wc_MlDsaKey *	**wc_MlDsaKey_New**(void * heap, int devId)Allocates and initializes a new wc_MlDsaKey on the heap. The returned pointer must be released with wc_MlDsaKey_Delete(). Only available when wolfSSL is built without WC_NO_CONSTRUCTORS.
int	**wc_MlDsaKey_Delete**(wc_MlDsaKey * key, wc_MlDsaKey ** key_p)Frees and zeros a heap_allocated wc_MlDsaKey previously returned by wc_MlDsaKey_New(). On success the caller's pointer variable is set to NULL via key_p when key_p is not NULL. Only available when wolfSSL is built without WC_NO_CONSTRUCTORS.

	Name
int	**wc_MlDsaKey_InitId**(wc_MlDsaKey * key, const unsigned char * id, int len, void * heap, int devId)Initializes a wc_MlDsaKey with a device_side key identifier. Equivalent to wc_MlDsaKey_Init() but also stashes a binary id that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.
int	**wc_MlDsaKey_InitLabel**(wc_MlDsaKey * key, const char * label, void * heap, int devId)Initializes a wc_MlDsaKey with a device_side key label. Equivalent to wc_MlDsaKey_Init() but also stashes a label string that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.
int	**wc_MlDsaKey_SetParams**(wc_MlDsaKey * key, byte level)Selects the ML_DSA parameter set for this key. Must be called after wc_MlDsaKey_Init() and before key generation, signing or verifying.
int	**wc_MlDsaKey_GetParams**(wc_MlDsaKey * key, byte * level)Retrieves the ML-DSA parameter set currently selected on this key.
void	**wc_MlDsaKey_Free**(wc_MlDsaKey * key)Releases resources held by a wc_MlDsaKey. After this call the object must be re_initialized with wc_MlDsaKey_Init() before it can be used again. Safe to call with a NULL pointer.
int	**wc_MlDsaKey_MakeKey**(wc_MlDsaKey * key, WC_RNG * rng)Generates a new ML_DSA key pair using the provided RNG. The parameter set must already be set with wc_MlDsaKey_SetParams(). Both the public and private key components are populated on success.
int	**wc_MlDsaKey_MakeKeyFromSeed**(wc_MlDsaKey * key, const byte * seed)Deterministically generates an ML-DSA key pair from a 32-byte seed. Useful for known-answer tests and for applications that derive the seed from another secret. The seed buffer must contain exactly 32 bytes.

	Name
int	**wc_MlDsaKey_MakePublicKey**(wc_MlDsaKey * key)Derives public key for a wc_MlDsaKey with private key set. If the public key is already set, it is only checked against the private key's tr (H(pk)); use wc_MlDsaKey_CheckKey() for a full key pair check. The private key is still required: a public-only key returns BAD_FUNC_ARG. Available when WC_MLDSA_HAVE_MAKE_PUBLIC_KEY is defined: that is, when WOLFSSL_MLDSA_ASSIGN_KEY, WOLFSSL_MLDSA_NO_MAKE_KEY and WOLFSSL_MLDSA_VERIFY_ONLY are not defined.
int	**wc_MlDsaKey_SignCtx**(wc_MlDsaKey * key, const byte * ctx, byte ctxLen, byte * sig, word32 * sigLen, const byte * msg, word32 msgLen, WC_RNG * rng)Signs a message with ML-DSA using the FIPS 204 randomized-with-context signing API. Pass ctx=NULL and ctxLen=0 for an empty context.
int	**wc_MlDsaKey_SignCtxHash**(wc_MlDsaKey * key, const byte * ctx, byte ctxLen, byte * sig, word32 * sigLen, const byte * hash, word32 hashLen, int hashAlg, WC_RNG * rng)HashML-DSA signing variant: signs a pre-hashed message. The caller supplies the hash bytes and identifies the hash algorithm. This is the "pre-hash" mode of FIPS 204.
int	**wc_MlDsaKey_Sign**(wc_MlDsaKey * key, byte * sig, word32 * sigLen, const byte * msg, word32 msgLen, WC_RNG * rng)Legacy ML_DSA signing API without a context parameter. Only available when wolfSSL is built with WOLFSSL_MLDSA_NO_CTX. New code should call wc_MlDsaKey_SignCtx() with ctx=NULL and ctxLen=0 for a FIPS 204 compliant empty-context signature.
int	**wc_MlDsaKey_SignCtxWithSeed**(wc_MlDsaKey * key, const byte * ctx, byte ctxLen, byte * sig, word32 * sigLen, const byte * msg, word32 msgLen, const byte * seed)Deterministic signing variant of wc_MlDsaKey_SignCtx(). The 32-byte seed replaces the randomness an RNG would supply, so the same key/ctx/msg/seed always produces the same signature.

	Name
int	**wc_MlDsaKey_SignCtxHashWithSeed**(wc_MlDsaKey * key, const byte * ctx, byte ctxLen, byte * sig, word32 * sigLen, const byte * hash, word32 hashLen, int hashAlg, const byte * seed)Deterministic HashML_DSA signing: like wc_MlDsaKey_SignCtxHash() but uses the supplied 32-byte seed in place of an RNG.
int	**wc_MlDsaKey_SignMuWithSeed**(wc_MlDsaKey * key, byte * sig, word32 * sigLen, const byte * mu, word32 muLen, const byte * seed)Signs a pre_computed mu value (the externally derived SHAKE256 hash of (tr
int	**wc_MlDsaKey_SignWithSeed**(wc_MlDsaKey * key, byte * sig, word32 * sigLen, const byte * msg, word32 msgLen, const byte * seed)Legacy seed_based signing API without a context parameter. Only available when wolfSSL is built with WOLFSSL_MLDSA_NO_CTX. New code should use wc_MlDsaKey_SignCtxWithSeed().
int	**wc_MlDsaKey_VerifyCtx**(wc_MlDsaKey * key, const byte * sig, word32 sigLen, const byte * ctx, byte ctxLen, const byte * msg, word32 msgLen, int * res)Verifies an ML_DSA signature produced by wc_MlDsaKey_SignCtx() or one of its variants. On entry res is set to 0; on success it is set to 1 when the signature is valid and left at 0 otherwise. The function's return value reports whether verification could be carried out at all; a bad signature is NOT a function-level error.
int	**wc_MlDsaKey_VerifyCtxHash**(wc_MlDsaKey * key, const byte * sig, word32 sigLen, const byte * ctx, byte ctxLen, const byte * hash, word32 hashLen, int hashAlg, int * res)Verifies a HashML_DSA signature where the message digest was supplied directly. See wc_MlDsaKey_VerifyCtx() for the semantics of res.
int	**wc_MlDsaKey_VerifyMu**(wc_MlDsaKey * key, const byte * sig, word32 sigLen, const byte * mu, word32 muLen, int * res)Verifies a signature over a pre_computed 64_byte mu value (see wc_MlDsaKey_SignMuWithSeed()). See wc_MlDsaKey_VerifyCtx() for the semantics of res.

	Name
int	**wc_MlDsaKey_SetPrecompA**(wc_MlDsaKey * key, const sword32 * a, word32 aLen, const byte * rho, word32 rhoLen)Attaches a matrix A that was expanded off target, so verification can skip the SHAKE128 rejection sampling that otherwise dominates it. A is a function of the public seed rho alone, so wherever the verification key is fixed at build time - secure boot being the usual case - A can be computed once on a host and stored in flash. Available only when WOLFSSL_MLDSA_VERIFY_PRECOMP_A is defined, and honoured by both the default and the small-memory verifiers.
int	**wc_MlDsaKey_Verify**(wc_MlDsaKey * key, const byte * sig, word32 sigLen, const byte * msg, word32 msgLen, int * res)Legacy ML_DSA verify API without a context parameter. Only available when wolfSSL is built with WOLFSSL_MLDSA_NO_CTX. New code should use wc_MlDsaKey_VerifyCtx() with ctx=NULL and ctxLen=0.
int	**wc_MlDsaKey_Size**(wc_MlDsaKey * key)Returns the size in bytes of the encoded private key for the parameter set selected on this key. Equivalent to wc_MlDsaKey_PrivSize() and provided for API compatibility.
int	**wc_MlDsaKey_PrivSize**(wc_MlDsaKey * key)Returns the size in bytes of the encoded private key for the parameter set selected on this key.
int	**wc_MlDsaKey_PubSize**(wc_MlDsaKey * key)Returns the size in bytes of the encoded public key for the parameter set selected on this key.
int	**wc_MlDsaKey_SigSize**(wc_MlDsaKey * key)Returns the size in bytes of the signature produced by this key's parameter set.
int	**wc_MlDsaKey_GetPrivLen**(wc_MlDsaKey * key, int * len)Writes the encoded private key size into *len. Equivalent information to wc_MlDsaKey_PrivSize() but uses an out-parameter style.
int	**wc_MlDsaKey_GetPubLen**(wc_MlDsaKey * key, int * len)Writes the encoded public key size into *len. Equivalent information to wc_MlDsaKey_PubSize() but uses an out-parameter style.

	Name
int	**wc_MlDsaKey_GetSigLen**(wc_MlDsaKey * key, int * len)Writes the signature size into *len. Equivalent information to wc_MlDsaKey_SigSize() but uses an out-parameter style.
int	**wc_MlDsaKey_CheckKey**(wc_MlDsaKey * key)Self-checks an ML-DSA key by recomputing the public key from the private and comparing against the stored public. Only available when wolfSSL is built with WOLFSSL_MLDSA_CHECK_KEY.
int	**wc_MlDsaKey_ImportPubRaw**(wc_MlDsaKey * key, const byte * in, word32 inLen)Imports a raw ML_DSA public key. The parameter set must already be selected on the key. inLen must match the size returned by wc_MlDsaKey_PubSize() for the selected parameter set.
int	**wc_MlDsaKey_ImportPrivRaw**(wc_MlDsaKey * key, const byte * priv, word32 privSz)Imports a raw ML_DSA private key. The parameter set must already be selected. privSz must match the size returned by wc_MlDsaKey_PrivSize().
int	**wc_MlDsaKey_ImportKey**(wc_MlDsaKey * key, const byte * priv, word32 privSz, const byte * pub, word32 pubSz)Imports a raw ML_DSA key pair (private and public parts together). The parameter set must already be selected.
int	**wc_MlDsaKey_ExportPubRaw**(wc_MlDsaKey * key, byte * out, word32 * outLen)Exports the raw ML-DSA public key. On entry *outLen is the size of out; on success it is updated to the bytes written.
int	**wc_MlDsaKey_ExportPrivRaw**(wc_MlDsaKey * key, byte * out, word32 * outLen)Exports the raw ML-DSA private key. On entry *outLen is the size of out; on success it is updated to the bytes written.
int	**wc_MlDsaKey_ExportKey**(wc_MlDsaKey * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz)Exports both raw public and private ML-DSA key components in one call.
int	**wc_MlDsaKey_PrivateKeyDecode**(wc_MlDsaKey * key, const byte * input, word32 inSz, word32 * inOutIdx)Parses an ML_DSA private key from a DER/ASN.1 encoded buffer (PKCS#8 OneAsymmetricKey). The parameter set is inferred from the algorithm identifier in the encoding, so it does NOT need to be set beforehand. On success *inOutIdx is advanced past the consumed bytes.

	Name
int	**wc_MlDsaKey_PublicKeyDecode**(wc_MlDsaKey * key, const byte * input, word32 inSz, word32 * inOutIdx)Parses an ML_DSA public key from a DER/ASN.1 encoded buffer (SubjectPublicKeyInfo). The parameter set is inferred from the algorithm identifier. On success *inOutIdx is advanced past the consumed bytes.
int	**wc_MlDsaKey_PublicKeyToDer**(wc_MlDsaKey * key, byte * output, word32 inLen, int withAlg)Encodes an ML_DSA public key to DER. When withAlg is non_zero the output is a full SubjectPublicKeyInfo (with AlgorithmIdentifier); when zero the output is the raw public key bytes.
int	**wc_MlDsaKey_KeyToDer**(wc_MlDsaKey * key, byte * output, word32 inLen)Encodes an ML_DSA key pair (public + private) to DER as a PKCS#8 OneAsymmetricKey structure. Pass NULL as output to query the required buffer size.
int	**wc_MlDsaKey_PrivateKeyToDer**(wc_MlDsaKey * key, byte * output, word32 inLen)Encodes the ML_DSA private key to DER. Per FIPS 204 the private key encoding includes the public component, so this function is currently an alias of wc_MlDsaKey_KeyToDer() kept for API parity with other algorithms.

B.39.2 Detailed Description

ML-DSA (Module-Lattice-based Digital Signature Algorithm) is a quantum-resistant digital signature scheme standardized by NIST as FIPS 204. The pre-standardization name was Dilithium; legacy Dilithium type and macro names remain as aliases for unmigrated consumer code (see <wolfssl/-wolfcrypt/dilithium.h>).

ML-DSA defines three parameter sets identified by NIST security category: ML-DSA-44 (level 2), ML-DSA-65 (level 3) and ML-DSA-87 (level 5). All three are supported by the same wc_MlDsaKey object; the parameter set is selected with wc_MlDsaKey_SetParams().

B.39.3 Functions Documentation

```
int wc_MlDsaKey_Init(
    wc_MlDsaKey * key,
    void * heap,
    int devId
)
```

Initializes a wc_MlDsaKey object. Must be called before any other ML-DSA operation. Use wc_MlDsaKey_Free() to release resources when done.

Parameters:

- **key** Pointer to the wc_MlDsaKey to initialize.
- **heap** Heap hint for dynamic memory allocation. May be NULL.
- **devId** Device identifier for hardware crypto callbacks. Use INVALID_DEVID for software-only.

See:

- wc_MlDsaKey_Free
- wc_MlDsaKey_SetParams
- wc_MlDsaKey_MakeKey

Return:

- 0 on success.
- BAD_FUNC_ARG if key is NULL.
- Other negative value if initializing the key's SHAKE-256 object fails.

ML-DSA (FIPS 204) is a quantum-resistant digital signature algorithm. Three parameter sets are defined and selected via wc_MlDsaKey_SetParams() after init:

- WC_ML_DSA_44 (NIST security level 2),
- WC_ML_DSA_65 (level 3),
- WC_ML_DSA_87 (level 5).

Example

```
wc_MlDsaKey key;
int ret;

ret = wc_MlDsaKey_Init(&key, NULL, INVALID_DEVID);
if (ret != 0) {
    // error initializing key
}
ret = wc_MlDsaKey_SetParams(&key, WC_ML_DSA_65);
// ... use key ...
wc_MlDsaKey_Free(&key);
```

```
wc_MlDsaKey * wc_MlDsaKey_New(
    void * heap,
    int devId
)
```

Allocates and initializes a new wc_MlDsaKey on the heap. The returned pointer must be released with wc_MlDsaKey_Delete(). Only available when wolfSSL is built without WC_NO_CONSTRUCTORS.

Parameters:

- **heap** Heap hint for dynamic memory allocation. May be NULL.
- **devId** Device identifier for hardware crypto callbacks. Use INVALID_DEVID for software-only.

See:

- wc_MlDsaKey_Delete
- wc_MlDsaKey_Init

Return:

- Pointer to a freshly allocated wc_MlDsaKey on success.
- NULL on allocation failure.

Example

```
wc_MlDsaKey* key = wc_MlDsaKey_New(NULL, INVALID_DEVID);
if (key == NULL) {
    // allocation failed
}
// ... use key ...
wc_MlDsaKey_Delete(key, &key);
```

```
int wc_MlDsaKey_Delete(
    wc_MlDsaKey * key,
    wc_MlDsaKey ** key_p
)
```

Frees and zeros a heap-allocated wc_MlDsaKey previously returned by wc_MlDsaKey_New(). On success the caller's pointer variable is set to NULL via key_p when key_p is not NULL. Only available when wolfSSL is built without WC_NO_CONSTRUCTORS.

Parameters:

- **key** The wc_MlDsaKey to free.
- **key_p** Optional address of the caller's pointer variable; when not NULL, it is set to NULL on success.

See: wc_MlDsaKey_New

Return:

- 0 on success.
- BAD_FUNC_ARG if key is NULL.

```
int wc_MlDsaKey_InitId(
    wc_MlDsaKey * key,
    const unsigned char * id,
    int len,
    void * heap,
    int devId
)
```

Initializes a wc_MlDsaKey with a device-side key identifier. Equivalent to wc_MlDsaKey_Init() but also stashes a binary id that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.

Parameters:

- **key** Pointer to the wc_MlDsaKey to initialize.
- **id** Pointer to the device-side key identifier bytes. May be NULL when len is 0.
- **len** Number of bytes in id; must be in [0, MLDSA_MAX_ID_LEN].
- **heap** Heap hint for dynamic memory allocation. May be NULL.
- **devId** Device identifier for the crypto callback. Should be a registered cb devId, not INVALID_DEVID.

See:

- wc_MlDsaKey_Init
- wc_MlDsaKey_InitLabel
- wc_MlDsaKey_Free

Return:

- 0 on success.
- BAD_FUNC_ARG if key is NULL, or if id is NULL and len is positive.
- BUFFER_E if len is negative or exceeds MLDSA_MAX_ID_LEN.

The id is copied into the key object; the caller may free its buffer immediately after this call returns.

The key is given the highest parameter set compiled in (WC_ML_DSA_87 unless it was disabled at build time), as the parameter set of the key held on the device is not known here. Call wc_MlDsaKey_SetParams() to select a different one.

```
int wc_MlDsaKey_InitLabel(
    wc_MlDsaKey * key,
    const char * label,
    void * heap,
    int devId
)
```

Initializes a wc_MlDsaKey with a device-side key label. Equivalent to wc_MlDsaKey_Init() but also stashes a label string that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.

Parameters:

- **key** Pointer to the wc_MlDsaKey to initialize.
- **label** NUL-terminated device-side key label string.
- **heap** Heap hint for dynamic memory allocation. May be NULL.
- **devId** Device identifier for the crypto callback.

See:

- wc_MlDsaKey_Init
- wc_MlDsaKey_InitId
- wc_MlDsaKey_Free

Return:

- 0 on success.
- BAD_FUNC_ARG if key or label is NULL.
- BUFFER_E if label is empty or longer than MLDSA_MAX_LABEL_LEN.

The label length is taken via XSTRLEN, so embedded NUL bytes terminate the label.

The parameter set is chosen as in wc_MlDsaKey_InitId().

```
int wc_MlDsaKey_SetParams(
    wc_MlDsaKey * key,
    byte level
)
```

Selects the ML-DSA parameter set for this key. Must be called after wc_MlDsaKey_Init() and before key generation, signing or verifying.

Parameters:

- **key** Pointer to an initialized wc_MlDsaKey.
- **level** Parameter set: WC_ML_DSA_44, WC_ML_DSA_65 or WC_ML_DSA_87.

See:

- wc_MlDsaKey_GetParams
- wc_MlDsaKey_Init

Return:

- 0 on success.
- BAD_FUNC_ARG if key is NULL or level is not a recognized parameter set.
- NOT_COMPILED_IN if level names a parameter set that was disabled at build time.

```
int wc_MlDsaKey_GetParams(
    wc_MlDsaKey * key,
    byte * level
)
```

Retrieves the ML-DSA parameter set currently selected on this key.

Parameters:

- **key** Pointer to an initialized wc_MlDsaKey.
- **level** Receives WC_ML_DSA_44, WC_ML_DSA_65 or WC_ML_DSA_87.

See: wc_MlDsaKey_SetParams

Return:

- 0 on success.
- BAD_FUNC_ARG if key or level is NULL.

```
void wc_MlDsaKey_Free(
    wc_MlDsaKey * key
)
```

Releases resources held by a wc_MlDsaKey. After this call the object must be re-initialized with wc_MlDsaKey_Init() before it can be used again. Safe to call with a NULL pointer.

Parameters:

- **key** Pointer to the wc_MlDsaKey to free.

See: wc_MlDsaKey_Init

```
int wc_MlDsaKey_MakeKey(
    wc_MlDsaKey * key,
    WC_RNG * rng
)
```

Generates a new ML-DSA key pair using the provided RNG. The parameter set must already be set with wc_MlDsaKey_SetParams(). Both the public and private key components are populated on success.

Parameters:

- **key** Pointer to a wc_MlDsaKey with parameters set.
- **rng** Pointer to an initialized WC_RNG.

See:

- wc_MlDsaKey_MakeKeyFromSeed
- wc_MlDsaKey_SetParams

Return:

- 0 on success.
- BAD_FUNC_ARG if key or rng is NULL.
- MEMORY_E on allocation failure.

Example

```
wc_MlDsaKey key;
WC_RNG rng;

wc_MlDsaKey_Init(&key, NULL, INVALID_DEVID);
wc_MlDsaKey_SetParams(&key, WC_ML_DSA_65);
wc_InitRng(&rng);

if (wc_MlDsaKey_MakeKey(&key, &rng) != 0) {
    // error generating key pair
}
```

```
int wc_MlDsaKey_MakeKeyFromSeed(
    wc_MlDsaKey * key,
    const byte * seed
)
```

Deterministically generates an ML-DSA key pair from a 32-byte seed. Useful for known-answer tests and for applications that derive the seed from another secret. The seed buffer must contain exactly 32 bytes.

Parameters:

- **key** Pointer to a wc_MlDsaKey with parameters set.
- **seed** Pointer to a 32-byte seed buffer.

See: wc_MlDsaKey_MakeKey

Return:

- 0 on success.
- BAD_FUNC_ARG if key or seed is NULL.

```
int wc_MlDsaKey_MakePublicKey(
    wc_MlDsaKey * key
)
```

Derives public key for a wc_MlDsaKey with private key set. If the public key is already set, it is only checked against the private key's tr (H(pk)); use wc_MlDsaKey_CheckKey() for a full key pair check. The private key is still required: a public-only key returns BAD_FUNC_ARG. Available when WC_MLDSA_HAVE_MAKE_PUBLIC_KEY is defined: that is, when WOLFSSL_MLDSA_ASSIGN_KEY, WOLFSSL_MLDSA_NO_MAKE_KEY and WOLFSSL_MLDSA_VERIFY_ONLY are not defined.

Parameters:

- **key** Pointer to wc_MlDsaKey.

See:

- wc_MlDsaKey_ImportPrivRaw

- wc_MlDsaKey_MakeKey
- wc_MlDsaKey_CheckKey

Return:

- 0 on success or already set.
- BAD_FUNC_ARG if invalid args or the private key is not set.
- MEMORY_E on allocation failure.
- PUBLIC_KEY_E if the public key does not match the private key's tr or, when derived, its t0.
- NO_VALID_DEVID with WOLF_CRYPTO_CB_ONLY_MLDSA.
- Other negative on error.

Derivation is always done in software from the local private key bytes, including for a key initialized with a devId: a key whose private key was imported with wc_MlDsaKey_ImportPrivRaw() (or generated in software) derives normally. A key that only references a private key held on a device (wc_MlDsaKey_InitId() / wc_MlDsaKey_InitLabel()) has no local private bytes, so it returns BAD_FUNC_ARG. On PUBLIC_KEY_E an already set public key is left in place; set the matching public key, or re-initialize the key, before deriving.

```
int wc_MlDsaKey_SignCtx(
    wc_MlDsaKey * key,
    const byte * ctx,
    byte ctxLen,
    byte * sig,
    word32 * sigLen,
    const byte * msg,
    word32 msgLen,
    WC_RNG * rng
)
```

Signs a message with ML-DSA using the FIPS 204 randomized-with-context signing API. Pass ctx=NULL and ctxLen=0 for an empty context.

Parameters:

- **key** Pointer to a wc_MlDsaKey with the private key.
- **ctx** Optional context string (may be NULL when ctxLen=0).
- **ctxLen** Length of ctx in bytes; must be 0 when ctx is NULL, and no greater than 255.
- **sig** Buffer that receives the signature.
- **sigLen** In: size of sig. Out: bytes written.
- **msg** Message to sign.
- **msgLen** Length of msg in bytes.
- **rng** Pointer to an initialized WC_RNG.

See:

- wc_MlDsaKey_VerifyCtx
- wc_MlDsaKey_SignCtxWithSeed
- wc_MlDsaKey_SignCtxHash

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL, ctxLen is invalid or the key has no private key set.
- BUFFER_E if the sig buffer is too small.

On entry *sigLen is the size of the sig buffer; on success it is updated to the number of bytes written. Use wc_MlDsaKey_SigSize() or wc_MlDsaKey_GetSigLen() to determine the required buffer size.

```
int wc_MlDsaKey_SignCtxHash(
    wc_MlDsaKey * key,
    const byte * ctx,
    byte ctxLen,
    byte * sig,
    word32 * sigLen,
    const byte * hash,
    word32 hashLen,
    int hashAlg,
    WC_RNG * rng
)
```

HashML-DSA signing variant: signs a pre-hashed message. The caller supplies the hash bytes and identifies the hash algorithm. This is the “pre-hash” mode of FIPS 204.

Parameters:

- **key** Pointer to a wc_MlDsaKey with the private key.
- **ctx** Optional context string (NULL when ctxLen=0).
- **ctxLen** Length of ctx in bytes; no greater than 255.
- **sig** Buffer that receives the signature.
- **sigLen** In: size of sig. Out: bytes written.
- **hash** The message digest to sign.
- **hashLen** Length of hash in bytes.
- **hashAlg** Hash algorithm identifier (e.g. WC_HASH_TYPE_SHA256).
- **rng** Pointer to an initialized WC_RNG.

See:

- wc_MlDsaKey_SignCtx
- wc_MlDsaKey_VerifyCtxHash

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL, ctxLen is invalid, hashAlg is not supported or is weaker than the parameter set (FIPS 204 sec 5.4), or the key has no private key set.
- BUFFER_E if the sig buffer is too small.

Note: WOLFSSL_MLDSA_ALLOW_WEAK_PREHASH accepts a supported hash weaker than the parameter set instead of returning BAD_FUNC_ARG. An unsupported hash is still refused. Not FIPS 204 section 5.4 compliant.

```
int wc_MlDsaKey_Sign(
    wc_MlDsaKey * key,
    byte * sig,
    word32 * sigLen,
    const byte * msg,
    word32 msgLen,
    WC_RNG * rng
)
```

Legacy ML-DSA signing API without a context parameter. Only available when wolfSSL is built with WOLFSSL_MLDSA_NO_CTX. New code should call wc_MlDsaKey_SignCtx() with ctx=NULL and ctxLen=0 for a FIPS 204 compliant empty-context signature.

Parameters:

- **key** Pointer to a wc_MlDsaKey with the private key.
- **sig** Buffer that receives the signature.
- **sigLen** In: size of sig. Out: bytes written.
- **msg** Message to sign.
- **msgLen** Length of msg in bytes.
- **rng** Pointer to an initialized WC_RNG.

See:

- wc_MlDsaKey_SignCtx
- wc_MlDsaKey_Verify

Return: See wc_MlDsaKey_SignCtx().

```
int wc_MlDsaKey_SignCtxWithSeed(
    wc_MlDsaKey * key,
    const byte * ctx,
    byte ctxLen,
    byte * sig,
    word32 * sigLen,
    const byte * msg,
    word32 msgLen,
    const byte * seed
)
```

Deterministic signing variant of wc_MlDsaKey_SignCtx(). The 32-byte seed replaces the randomness an RNG would supply, so the same key/ctx/msg/seed always produces the same signature.

Parameters:

- **key** Pointer to a wc_MlDsaKey with the private key.
- **ctx** Optional context string (NULL when ctxLen=0).
- **ctxLen** Length of ctx; no greater than 255.
- **sig** Buffer that receives the signature.

- **sigLen** In: size of sig. Out: bytes written.
- **msg** Message to sign.
- **msgLen** Length of msg in bytes.
- **seed** 32-byte seed bytes.

See:

- wc_MlDsaKey_SignCtx
- wc_MlDsaKey_SignCtxHashWithSeed

Return: See wc_MlDsaKey_SignCtx().

```
int wc_MlDsaKey_SignCtxHashWithSeed(
    wc_MlDsaKey * key,
    const byte * ctx,
    byte ctxLen,
    byte * sig,
    word32 * sigLen,
    const byte * hash,
    word32 hashLen,
    int hashAlg,
    const byte * seed
)
```

Deterministic HashML-DSA signing: like wc_MlDsaKey_SignCtxHash() but uses the supplied 32-byte seed in place of an RNG.

Parameters:

- **key** Pointer to a wc_MlDsaKey with the private key.
- **ctx** Optional context string (NULL when ctxLen=0).
- **ctxLen** Length of ctx; no greater than 255.
- **sig** Buffer that receives the signature.
- **sigLen** In: size of sig. Out: bytes written.
- **hash** The message digest to sign.
- **hashLen** Length of hash in bytes.
- **hashAlg** Hash algorithm identifier.
- **seed** 32-byte seed bytes.

See: wc_MlDsaKey_SignCtxHash

Return: See wc_MlDsaKey_SignCtxHash().

```
int wc_MlDsaKey_SignMuWithSeed(
    wc_MlDsaKey * key,
    byte * sig,
    word32 * sigLen,
    const byte * mu,
    word32 muLen,
    const byte * seed
)
```

Signs a pre-computed mu value (the externally derived SHAKE256 hash of (tr || ctx || msg) per FIPS 204) using a deterministic 32-byte seed. Used by protocols that need to split the message-hashing step from the signing step.

Parameters:

- **key** Pointer to a wc_MlDsaKey with the private key.
- **sig** Buffer that receives the signature.
- **sigLen** In: size of sig. Out: bytes written.
- **mu** The 64-byte mu value (SHAKE256 output).
- **muLen** Length of mu; must be 64.
- **seed** 32-byte seed bytes.

See: wc_MlDsaKey_VerifyMu

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL, muLen is not 64 or the key has no private key set.
- BUFFER_E if the sig buffer is too small.

```
int wc_MlDsaKey_SignWithSeed(
    wc_MlDsaKey * key,
    byte * sig,
    word32 * sigLen,
    const byte * msg,
    word32 msgLen,
    const byte * seed
)
```

Legacy seed-based signing API without a context parameter. Only available when wolfSSL is built with WOLFSSL_MLDSA_NO_CTX. New code should use wc_MlDsaKey_SignCtxWithSeed().

Parameters:

- **key** Pointer to a wc_MlDsaKey with the private key.
- **sig** Buffer that receives the signature.
- **sigLen** In: size of sig. Out: bytes written.
- **msg** Message to sign.
- **msgLen** Length of msg in bytes.
- **seed** 32-byte seed bytes.

See: wc_MlDsaKey_SignCtxWithSeed

Return: See wc_MlDsaKey_SignCtxWithSeed().

```
int wc_MlDsaKey_VerifyCtx(
    wc_MlDsaKey * key,
    const byte * sig,
```

```
    word32 sigLen,
    const byte * ctx,
    byte ctxLen,
    const byte * msg,
    word32 msgLen,
    int * res
)
```

Verifies an ML-DSA signature produced by wc_MlDsaKey_SignCtx() or one of its variants. On entry res is set to 0; on success it is set to 1 when the signature is valid and left at 0 otherwise. The function's return value reports whether verification could be carried out at all; a bad signature is NOT a function-level error.

Parameters:

- **key** Pointer to a wc_MlDsaKey with the public key.
- **sig** Signature bytes to verify.
- **sigLen** Length of sig in bytes.
- **ctx** Optional context string (NULL when ctxLen=0).
- **ctxLen** Length of ctx; no greater than 255.
- **msg** Message that was signed.
- **msgLen** Length of msg in bytes.
- **res** Set to 1 on a valid signature, 0 otherwise.

See:

- wc_MlDsaKey_SignCtx
- wc_MlDsaKey_VerifyCtxHash
- wc_MlDsaKey_VerifyMu

Return:

- 0 if verification completed (check res for the result).
- BAD_FUNC_ARG if any required pointer is NULL or ctxLen is invalid.

```
int wc_MlDsaKey_VerifyCtxHash(
    wc_MlDsaKey * key,
    const byte * sig,
    word32 sigLen,
    const byte * ctx,
    byte ctxLen,
    const byte * hash,
    word32 hashLen,
    int hashAlg,
    int * res
)
```

Verifies a HashML-DSA signature where the message digest was supplied directly. See wc_MlDsaKey_VerifyCtx() for the semantics of res.

Parameters:

- **key** Pointer to a wc_MlDsaKey with the public key.
- **sig** Signature bytes to verify.
- **sigLen** Length of sig in bytes.
- **ctx** Optional context string (NULL when ctxLen=0).
- **ctxLen** Length of ctx; no greater than 255.
- **hash** The message digest that was signed.
- **hashLen** Length of hash in bytes.
- **hashAlg** Hash algorithm identifier.
- **res** Set to 1 on a valid signature, 0 otherwise.

See:

- wc_MlDsaKey_SignCtxHash
- wc_MlDsaKey_VerifyCtx

Return:

- 0 if verification completed (check res for the result).
- BAD_FUNC_ARG if any required pointer is NULL, ctxLen is invalid, or hashAlg is unsupported or weaker than the parameter set (FIPS 204 sec 5.4).

Note: WOLFSSL_MLDSA_ALLOW_WEAK_PREHASH verifies such a signature rather than refusing it. An unsupported hash is still refused.

```
int wc_MlDsaKey_VerifyMu(
    wc_MlDsaKey * key,
    const byte * sig,
    word32 sigLen,
    const byte * mu,
    word32 muLen,
    int * res
)
```

Verifies a signature over a pre-computed 64-byte mu value (see wc_MlDsaKey_SignMuWithSeed()). See wc_MlDsaKey_VerifyCtx() for the semantics of res.

Parameters:

- **key** Pointer to a wc_MlDsaKey with the public key.
- **sig** Signature bytes to verify.
- **sigLen** Length of sig in bytes.
- **mu** The 64-byte mu value.
- **muLen** Length of mu; must be 64.
- **res** Set to 1 on a valid signature, 0 otherwise.

See: wc_MlDsaKey_SignMuWithSeed

Return:

- 0 if verification completed (check res for the result).

- BAD_FUNC_ARG if any required pointer is NULL or muLen is not 64.

```
int wc_MlDsaKey_SetPrecompA(
    wc_MlDsaKey * key,
    const sword32 * a,
    word32 aLen,
    const byte * rho,
    word32 rhoLen
)
```

Attaches a matrix A that was expanded off target, so verification can skip the SHAKE128 rejection sampling that otherwise dominates it. A is a function of the public seed rho alone, so wherever the verification key is fixed at build time - secure boot being the usual case - A can be computed once on a host and stored in flash. Available only when WOLFSSL_MLDSA_VERIFY_PRECOMP_A is defined, and honoured by both the default and the small-memory verifiers.

Parameters:

- **key** Pointer to a wc_MlDsaKey holding the public key.
- **a** Expanded matrix A, k * l * MLDSA_N elements, caller owned.
- **aLen** Number of sword32 elements in a.
- **rho** Public seed the matrix was expanded from.
- **rhoLen** Length of rho; must be MLDSA_PUB_SEED_SZ.

See:

- wc_MlDsaKey_VerifyMu
- wc_MlDsaKey_ImportPubRaw

Return:

- 0 on success.
- BAD_FUNC_ARG if a pointer is NULL, rhoLen is not MLDSA_PUB_SEED_SZ, aLen is not k * l * MLDSA_N, or no public key has been imported yet.
- PUBLIC_KEY_E if rho does not match the imported public key.

Note: A is derived from the public key and must be integrity-protected exactly as the public key is. An attacker able to substitute it can influence verification. In a secure-boot design the key already lives in protected flash, so store the two together.

The matrix is borrowed, never copied and never freed: it must stay valid and unchanged for as long as the key uses it. It holds k * l * MLDSA_N sword32 elements (not bytes) in row-major (r, s) order and in the NTT domain, exactly as ExpandA produces them.

Import the public key first: the matrix is bound to it by comparing rho. The binding is re-checked on every verify, so replacing the key or the parameters afterwards simply discards the matrix and falls back to expanding A rather than using a stale one.

Example

```
wc_MlDsaKey key;
int res = 0;

wc_MlDsaKey_Init(&key, NULL, INVALID_DEVID);
wc_MlDsaKey_SetParams(&key, WC_ML_DSA_87);
wc_MlDsaKey_ImportPubRaw(&key, pub, pubLen);
if (wc_MlDsaKey_SetPrecompA(&key, matrixA, matrixALen, pub,
        MLDSA_PUB_SEED_SZ) != 0) {
    // handle error
}
wc_MlDsaKey_VerifyMu(&key, sig, sigLen, mu, muLen, &res);
```

```
int wc_MlDsaKey_Verify(
    wc_MlDsaKey * key,
    const byte * sig,
    word32 sigLen,
    const byte * msg,
    word32 msgLen,
    int * res
)
```

Legacy ML-DSA verify API without a context parameter. Only available when wolfSSL is built with WOLFSSL_MLDSA_NO_CTX. New code should use wc_MlDsaKey_VerifyCtx() with ctx=NULL and ctxLen=0.

Parameters:

- **key** Pointer to a wc_MlDsaKey with the public key.
- **sig** Signature bytes to verify.
- **sigLen** Length of sig in bytes.
- **msg** Message that was signed.
- **msgLen** Length of msg in bytes.
- **res** Set to 1 on a valid signature, 0 otherwise.

See:

- wc_MlDsaKey_VerifyCtx
- wc_MlDsaKey_Sign

Return: See wc_MlDsaKey_VerifyCtx().

```
int wc_MlDsaKey_Size(
    wc_MlDsaKey * key
)
```

Returns the size in bytes of the encoded private key for the parameter set selected on this key. Equivalent to wc_MlDsaKey_PrivSize() and provided for API compatibility.

Parameters:

- **key** Pointer to a wc_MlDsaKey with parameters set.

See:

- wc_MlDsaKey_PrivSize
- wc_MlDsaKey_PubSize
- wc_MlDsaKey_SigSize

Return:

- Encoded private key size in bytes on success (positive value).
- BAD_FUNC_ARG if key is NULL or no parameter set is selected.

```
int wc_MlDsaKey_PrivSize(
    wc_MlDsaKey * key
)
```

Returns the size in bytes of the encoded private key for the parameter set selected on this key.

Parameters:

- **key** Pointer to a wc_MlDsaKey with parameters set.

See:

- wc_MlDsaKey_PubSize
- wc_MlDsaKey_GetPrivLen

Return:

- Encoded private key size on success (positive value).
- BAD_FUNC_ARG if key is NULL or no parameter set is selected.

```
int wc_MlDsaKey_PubSize(
    wc_MlDsaKey * key
)
```

Returns the size in bytes of the encoded public key for the parameter set selected on this key.

Parameters:

- **key** Pointer to a wc_MlDsaKey with parameters set.

See:

- wc_MlDsaKey_PrivSize
- wc_MlDsaKey_GetPubLen

Return:

- Encoded public key size on success (positive value).
- BAD_FUNC_ARG if key is NULL or no parameter set is selected.

```
int wc_MlDsaKey_SigSize(
    wc_MlDsaKey * key
)
```

Returns the size in bytes of the signature produced by this key's parameter set.

Parameters:

- **key** Pointer to a wc_MlDsaKey with parameters set.

See:

- wc_MlDsaKey_GetSigLen
- wc_MlDsaKey_SignCtx

Return:

- Signature size on success (positive value).
- BAD_FUNC_ARG if key is NULL or no parameter set is selected.

```
int wc_MlDsaKey_GetPrivLen(
    wc_MlDsaKey * key,
    int * len
)
```

Writes the encoded private key size into *len. Equivalent information to wc_MlDsaKey_PrivSize() but uses an out-parameter style.

Parameters:

- **key** Pointer to a wc_MlDsaKey with parameters set.
- **len** Receives the private key size in bytes.

See: wc_MlDsaKey_PrivSize

Return:

- 0 on success.
- BAD_FUNC_ARG if key or len is NULL, or no parameter set is selected.

```
int wc_MlDsaKey_GetPubLen(
    wc_MlDsaKey * key,
    int * len
)
```

Writes the encoded public key size into *len. Equivalent information to wc_MlDsaKey_PubSize() but uses an out-parameter style.

Parameters:

- **key** Pointer to a wc_MlDsaKey with parameters set.
- **len** Receives the public key size in bytes.

See: wc_MlDsaKey_PubSize

Return:

- 0 on success.
- BAD_FUNC_ARG if key or len is NULL, or no parameter set is selected.

```
int wc_MlDsaKey_GetSigLen(
    wc_MlDsaKey * key,
    int * len
)
```

Writes the signature size into *len. Equivalent information to wc_MlDsaKey_SigSize() but uses an out-parameter style.

Parameters:

- **key** Pointer to a wc_MlDsaKey with parameters set.
- **len** Receives the signature size in bytes.

See: wc_MlDsaKey_SigSize

Return:

- 0 on success.
- BAD_FUNC_ARG if key or len is NULL, or no parameter set is selected.

```
int wc_MlDsaKey_CheckKey(
    wc_MlDsaKey * key
)
```

Self-checks an ML-DSA key by recomputing the public key from the private and comparing against the stored public. Only available when wolfSSL is built with WOLFSSL_MLDSA_CHECK_KEY.

Parameters:

- **key** Pointer to a wc_MlDsaKey with both public and private parts populated.

Return:

- 0 if the key is consistent.
- BAD_FUNC_ARG if key is NULL.
- PUBLIC_KEY_E if the recomputed public does not match.

```
int wc_MlDsaKey_ImportPubRaw(
    wc_MlDsaKey * key,
    const byte * in,
    word32 inLen
)
```

Imports a raw ML-DSA public key. The parameter set must already be selected on the key. inLen must match the size returned by wc_MlDsaKey_PubSize() for the selected parameter set.

Parameters:

- **key** Pointer to a wc_MlDsaKey with parameters set.
- **in** Raw public key bytes.
- **inLen** Length of in in bytes.

See:

- wc_MlDsaKey_ExportPubRaw
- wc_MlDsaKey_ImportPrivRaw

Return:

- 0 on success.
- BAD_FUNC_ARG if key or in is NULL.
- BUFFER_E if inLen does not match the expected public key size.

```
int wc_MlDsaKey_ImportPrivRaw(
    wc_MlDsaKey * key,
    const byte * priv,
    word32 privSz
)
```

Imports a raw ML-DSA private key. The parameter set must already be selected. privSz must match the size returned by wc_MlDsaKey_PrivSize().

Parameters:

- **key** Pointer to a wc_MlDsaKey with parameters set.
- **priv** Raw private key bytes.
- **privSz** Length of priv in bytes.

See:

- wc_MlDsaKey_ExportPrivRaw
- wc_MlDsaKey_ImportKey

Return:

- 0 on success.
- BAD_FUNC_ARG if key or priv is NULL.
- BUFFER_E if privSz does not match the expected private key size.

```
int wc_MlDsaKey_ImportKey(
    wc_MlDsaKey * key,
    const byte * priv,
    word32 privSz,
    const byte * pub,
    word32 pubSz
)
```

Imports a raw ML-DSA key pair (private and public parts together). The parameter set must already be selected.

Parameters:

- **key** Pointer to a wc_MlDsaKey with parameters set.
- **priv** Raw private key bytes.
- **privSz** Length of priv.
- **pub** Raw public key bytes.
- **pubSz** Length of pub.

See: wc_MlDsaKey_ExportKey

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- BUFFER_E if privSz or pubSz does not match the expected sizes.

```
int wc_MlDsaKey_ExportPubRaw(
    wc_MlDsaKey * key,
    byte * out,
    word32 * outLen
)
```

Exports the raw ML-DSA public key. On entry *outLen is the size of out; on success it is updated to the bytes written.

Parameters:

- **key** Pointer to a wc_MlDsaKey with a public key.
- **out** Buffer that receives the public key.
- **outLen** In: size of out. Out: bytes written.

See: wc_MlDsaKey_ImportPubRaw

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- BUFFER_E if *outLen is smaller than the public key size.

```
int wc_MlDsaKey_ExportPrivRaw(
    wc_MlDsaKey * key,
    byte * out,
    word32 * outLen
)
```

Exports the raw ML-DSA private key. On entry *outLen is the size of out; on success it is updated to the bytes written.

Parameters:

- **key** Pointer to a wc_MlDsaKey with a private key.
- **out** Buffer that receives the private key.
- **outLen** In: size of out. Out: bytes written.

See: wc_MlDsaKey_ImportPrivRaw

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- BUFFER_E if *outLen is smaller than the private key size.

```
int wc_MlDsaKey_ExportKey(
    wc_MlDsaKey * key,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz
)
```

Exports both raw public and private ML-DSA key components in one call.

Parameters:

- **key** Pointer to a wc_MlDsaKey with both key parts.
- **priv** Buffer that receives the private key.

- **privSz** In: size of priv. Out: bytes written.
- **pub** Buffer that receives the public key.
- **pubSz** In: size of pub. Out: bytes written.

See: wc_MlDsaKey_ImportKey

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- BUFFER_E if either buffer is too small.

```
int wc_MlDsaKey_PrivateKeyDecode(
    wc_MlDsaKey * key,
    const byte * input,
    word32 inSz,
    word32 * inOutIdx
)
```

Parses an ML-DSA private key from a DER/ASN.1 encoded buffer (PKCS#8 OneAsymmetricKey). The parameter set is inferred from the algorithm identifier in the encoding, so it does NOT need to be set beforehand. On success *inOutIdx is advanced past the consumed bytes.

Parameters:

- **key** Pointer to an initialized wc_MlDsaKey.
- **input** DER-encoded private key bytes.
- **inSz** Length of input in bytes.
- **inOutIdx** In: offset into input where decoding starts. Out: offset past the consumed bytes.

See:

- wc_MlDsaKey_PrivateKeyToDer
- wc_MlDsaKey_PublicKeyDecode

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- ASN_PARSE_E on malformed encoding.

Only available when WOLFSSL_MLDSA_NO_ASN1 is not defined.

```
int wc_MlDsaKey_PublicKeyDecode(
    wc_MlDsaKey * key,
    const byte * input,
    word32 inSz,
    word32 * inOutIdx
)
```

Parses an ML-DSA public key from a DER/ASN.1 encoded buffer (SubjectPublicKeyInfo). The parameter set is inferred from the algorithm identifier. On success *inOutIdx is advanced past the consumed bytes.

Parameters:

- **key** Pointer to an initialized wc_MlDsaKey.
- **input** DER-encoded SPKI bytes.
- **inSz** Length of input in bytes.
- **inOutIdx** In: offset into input where decoding starts. Out: offset past the consumed bytes.

See: wc_MlDsaKey_PublicKeyToDer

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- ASN_PARSE_E on malformed encoding.

```
int wc_MlDsaKey_PublicKeyToDer(
    wc_MlDsaKey * key,
    byte * output,
    word32 inLen,
    int withAlg
)
```

Encodes an ML-DSA public key to DER. When withAlg is non-zero the output is a full SubjectPublicKeyInfo (with AlgorithmIdentifier); when zero the output is the raw public key bytes.

Parameters:

- **key** Pointer to a wc_MlDsaKey with a public key.
- **output** Buffer that receives the DER encoding, or NULL to query size.
- **inLen** Size of output (ignored when output is NULL).
- **withAlg** Non-zero to emit SubjectPublicKeyInfo, zero for the raw public key only.

See:

- wc_MlDsaKey_PublicKeyDecode
- wc_MlDsaKey_KeyToDer

Return:

- Size of the encoded DER in bytes on success.
- BAD_FUNC_ARG if key is NULL or no parameter set is selected.
- BUFFER_E if output is non-NULL and inLen is smaller than the required size.

Pass NULL as output to query the required buffer size.

```
int wc_MlDsaKey_KeyToDer(
    wc_MlDsaKey * key,
    byte * output,
    word32 inLen
)
```

Encodes an ML-DSA key pair (public + private) to DER as a PKCS#8 OneAsymmetricKey structure. Pass NULL as output to query the required buffer size.

Parameters:

- **key** Pointer to a wc_MlDsaKey with the private key.
- **output** Buffer that receives the DER encoding, or NULL to query size.
- **inLen** Size of output (ignored when output is NULL).

See:

- wc_MlDsaKey_PrivateKeyDecode
- wc_MlDsaKey_PrivateKeyToDer
- wc_MlDsaKey_PublicKeyToDer

Return:

- Size of the encoded DER in bytes on success.
- BAD_FUNC_ARG if key is NULL or no parameter set is selected.
- MISSING_KEY if the private key has not been set.
- BUFFER_E if output is non-NULL and inLen is too small.

```
int wc_MlDsaKey_PrivateKeyToDer(
    wc_MlDsaKey * key,
    byte * output,
    word32 inLen
)
```

Encodes the ML-DSA private key to DER. Per FIPS 204 the private key encoding includes the public component, so this function is currently an alias of wc_MlDsaKey_KeyToDer() kept for API parity with other algorithms.

Parameters:

- **key** Pointer to a wc_MlDsaKey with the private key.
- **output** Buffer that receives the DER encoding, or NULL to query size.
- **inLen** Size of output (ignored when output is NULL).

See:

- wc_MlDsaKey_KeyToDer
- wc_MlDsaKey_PrivateKeyDecode

Return:

- Size of the encoded DER in bytes on success.
- Inherited error codes from wc_MlDsaKey_KeyToDer().

B.40 Algorithms - ML-KEM (FIPS 203)

B.39.3.42 function wc_MlDsaKey_PrivateKeyToDer More...

B.40.1 Functions

	Name
MlKemKey *	**wc_MlKemKey_New**(int type, void * heap, int devId)Allocates and initializes a new MlKemKey on the heap. The returned pointer must be released with wc_MlKemKey_Delete().
int	**wc_MlKemKey_Delete**(MlKemKey * key, MlKemKey ** key_p)Frees and zeros a heap_allocated MlKemKey previously returned by wc_MlKemKey_New(). On success the caller's pointer variable is set to NULL via key_p when key_p is not NULL.
int	**wc_MlKemKey_Init**(MlKemKey * key, int type, void * heap, int devId)Initializes an MlKemKey object in place. The type parameter selects the ML-KEM variant and must be one of WC_ML_KEM_512, WC_ML_KEM_768 or WC_ML_KEM_1024.
int	**wc_MlKemKey_Free**(MlKemKey * key)Releases resources held by an MlKemKey. After this call the object must be re_initialized with wc_MlKemKey_Init() before it can be used again. Safe to call with a NULL pointer.
int	**wc_MlKemKey_Init_Id**(MlKemKey * key, int type, const unsigned char * id, int len, void * heap, int devId)Initializes an MlKemKey with a device_side key identifier. Equivalent to wc_MlKemKey_Init() but also stashes a binary id that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.

	Name
int	**wc_MlKemKey_Init_Label**(MlKemKey * key, int type, const char * label, void * heap, int devId)Initializes an MlKemKey with a device_side key label. Equivalent to wc_MlKemKey_Init() but also stashes a label string that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.
int	**wc_MlKemKey_MakeKey**(MlKemKey * key, WC_RNG * rng)Generates a new ML-KEM key pair using the provided RNG. Both the public and private components are populated on success.
int	**wc_MlKemKey_MakeKeyWithRandom**(MlKemKey * key, const unsigned char * rand, int len)Deterministic key generation: produces an ML-KEM key pair from the supplied 64 bytes of randomness instead of an RNG. Useful for known-answer tests and for applications that derive key randomness from another secret.
int	**wc_MlKemKey_CipherTextSize**(MlKemKey * key, word32 * len)Returns the ciphertext size in bytes for the variant selected on this key.
int	**wc_MlKemKey_SharedSecretSize**(MlKemKey * key, word32 * len)Returns the shared_secret size in bytes for ML_KEM. The value is the same (32 bytes) across all parameter sets but is queried programmatically for symmetry with wc_MlKemKey_CipherTextSize().
int	**wc_MlKemKey_Encapsulate**(MlKemKey * key, unsigned char * ct, unsigned char * ss, WC_RNG * rng)Encapsulates a fresh shared secret against the public key held in key. Produces a ciphertext that the holder of the corresponding private key can pass to wc_MlKemKey_Decapsulate() to recover the same shared secret.
int	**wc_MlKemKey_EncapsulateWithRandom**(MlKemKey * key, unsigned char * ct, unsigned char * ss, const unsigned char * rand, int len)Deterministic variant of wc_MlKemKey_Encapsulate(). Uses the supplied 32 bytes of randomness instead of consuming output from an RNG.

	Name
int	**wc_MlKemKey_Decapsulate**(MlKemKey * key, unsigned char * ss, const unsigned char * ct, word32 len)Decapsulates a ciphertext using the private key held in key and recovers the shared secret produced by wc_MlKemKey_Encapsulate(). ML_KEM decapsulation is constant time and includes an implicit_rejection check on malformed ciphertexts (an attacker cannot learn the validity of ct from the runtime).
int	**wc_MlKemKey_DecodePrivateKey**(MlKemKey * key, const unsigned char * in, word32 len)Decodes a raw ML_KEM private key into key. The variant must already be selected on the key (typically via wc_MlKemKey_Init() or wc_MlKemKey_New()) and len must match the private key size for that variant.
int	**wc_MlKemKey_DecodePublicKey**(MlKemKey * key, const unsigned char * in, word32 len)Decodes a raw ML-KEM public key into key. The variant must already be selected on the key and len must match the public key size for that variant.
int	**wc_MlKemKey_PrivateKeySize**(MlKemKey * key, word32 * len)Returns the encoded private key size in bytes for the variant selected on this key.
int	**wc_MlKemKey_PublicKeySize**(MlKemKey * key, word32 * len)Returns the encoded public key size in bytes for the variant selected on this key.
int	**wc_MlKemKey_EncodePrivateKey**(MlKemKey * key, unsigned char * out, word32 len)Encodes the ML_KEM private key into out. The out buffer length must be exactly wc_MlKemKey_PrivateKeySize() bytes.
int	**wc_MlKemKey_EncodePublicKey**(MlKemKey * key, unsigned char * out, word32 len)Encodes the ML_KEM public key into out. The out buffer length must be exactly wc_MlKemKey_PublicKeySize() bytes.
int	**wc_MlKemKey_PublicKeyToDer**(MlKemKey * key, byte * output, word32 len, int withAlg)Encodes the ML-KEM public key as a DER SubjectPublicKeyInfo. Pass NULL for output to obtain the required length.
int	**wc_MlKemKey_PrivateKeyToDer**(MlKemKey * key, byte * output, word32 len)Encodes the ML-KEM private key as a DER PKCS#8 OneAsymmetricKey, carrying the expanded decapsulation key. Pass NULL for output to obtain the required length.

	Name
int	**wc_MlKemKey_PublicKeyDecode**(MlKemKey * key, const byte * input, word32 inSz, word32 * inOutIdx)Decodes a DER SubjectPublicKeyInfo into an ML-KEM public key. Takes wrapped DER, where wc_MlKemKey_DecodePublicKey takes the raw encoded key. A key initialized for a parameter set holds the DER to that same one. A key initialized with WC_ML_KEM_TYPE_UNSET takes the parameter set from the algorithm OID in the DER instead.
int	**wc_MlKemKey_PrivateKeyDecode**(MlKemKey * key, const byte * input, word32 inSz, word32 * inOutIdx)Decodes a DER PKCS#8 OneAsymmetricKey into an ML_KEM private key. Takes wrapped DER, where wc_MlKemKey_DecodePrivateKey takes the raw encoded key. All three RFC 9935 Section 6 CHOICE forms are accepted: the 64_byte seed under an implicit [0], the expanded decapsulation key as an OCTET STRING, and the SEQUENCE carrying both. A seed is expanded with ML_KEM.KeyGen_internal(d,z). For the “both” form the expanded key is regenerated from the seed and the two are compared, so a key whose halves disagree is rejected as malformed, as RFC 9935 Section 8 requires. A key initialized for a parameter set holds the DER to that same parameter set; a key initialized with WC_ML_KEM_TYPE_UNSET takes it from the algorithm OID in the DER instead.

B.40.2 Detailed Description

ML-KEM (Module-Lattice-based Key Encapsulation Mechanism) is a quantum-resistant key encapsulation mechanism standardized by NIST as FIPS 203. The pre-standardization name was Kyber; legacy Kyber type and macro names remain as aliases for unmigrated consumer code.

ML-KEM defines three parameter sets: ML-KEM-512 (NIST level 1), ML-KEM-768 (level 3) and ML-KEM-1024 (level 5). The variant is selected when the key is initialized via wc_MlKemKey_Init() or wc_MlKemKey_New().

B.40.3 Functions Documentation

```
MlKemKey * wc_MlKemKey_New(
    int type,
    void * heap,
    int devId
)
```

Allocates and initializes a new MlKemKey on the heap. The returned pointer must be released with wc_MlKemKey_Delete().

Parameters:

- **type** ML-KEM variant: WC_ML_KEM_512, WC_ML_KEM_768 or WC_ML_KEM_1024.
- **heap** Heap hint for dynamic memory allocation. May be NULL.
- **devId** Device identifier for hardware crypto callbacks. Use INVALID_DEVID for software-only.

See:

- wc_MlKemKey_Delete
- wc_MlKemKey_Init

Return:

- Pointer to a freshly allocated MlKemKey on success.
- NULL on allocation failure or if type is invalid.

ML-KEM (FIPS 203) is a quantum-resistant key encapsulation mechanism. The type parameter selects the variant: WC_ML_KEM_512 (NIST security level 1), WC_ML_KEM_768 (level 3) or WC_ML_KEM_1024 (level 5).

Example

```
MlKemKey* key = wc_MlKemKey_New(WC_ML_KEM_768, NULL,
    INVALID_DEVID);
if (key == NULL) {
    // allocation failed
}
// ... use key ...
wc_MlKemKey_Delete(key, &key);
```

```
int wc_MlKemKey_Delete(
    MlKemKey * key,
    MlKemKey ** key_p
)
```

Frees and zeros a heap-allocated MlKemKey previously returned by wc_MlKemKey_New(). On success the caller's pointer variable is set to NULL via key_p when key_p is not NULL.

Parameters:

- **key** The MlKemKey to free.
- **key_p** Optional address of the caller's pointer variable; when not NULL, it is set to NULL on success.

See: wc_MlKemKey_New

Return:

- 0 on success.
- BAD_FUNC_ARG if key is NULL.

```
int wc_MlKemKey_Init(
    MlKemKey * key,
    int type,
    void * heap,
    int devId
)
```

Initializes an MlKemKey object in place. The type parameter selects the ML-KEM variant and must be one of WC_ML_KEM_512, WC_ML_KEM_768 or WC_ML_KEM_1024.

Parameters:

- **key** Pointer to the MlKemKey to initialize.
- **type** ML-KEM variant: WC_ML_KEM_512, WC_ML_KEM_768 or WC_ML_KEM_1024. Pass WC_ML_KEM_TYPE_UNSET to set the object up without a parameter set, leaving wc_MlKemKey_PublicKeyDecode or wc_MlKemKey_PrivateKeyDecode to take it from the algorithm OID in the DER. Every other operation needs a parameter set and fails with BAD_FUNC_ARG until a decode has named one.
- **heap** Heap hint for dynamic memory allocation. May be NULL.
- **devId** Device identifier for hardware crypto callbacks.

See:

- wc_MlKemKey_Free
- wc_MlKemKey_MakeKey

Return:

- 0 on success.
- BAD_FUNC_ARG if key is NULL or type is invalid.
- NOT_COMPILED_IN if type names a variant that was disabled at build time.

Example

```
MlKemKey key;
int ret;

ret = wc_MlKemKey_Init(&key, WC_ML_KEM_768, NULL, INVALID_DEVID);
if (ret != 0) {
    // error initializing key
}
// ... use key ...
wc_MlKemKey_Free(&key);
```

```
int wc_MlKemKey_Free(
    MlKemKey * key
)
```

Releases resources held by an MlKemKey. After this call the object must be re-initialized with wc_MlKemKey_Init() before it can be used again. Safe to call with a NULL pointer.

Parameters:

- **key** Pointer to the MlKemKey to free.

See: wc_MlKemKey_Init

Return: 0 on success, including when key is NULL.

```
int wc_MlKemKey_Init_Id(
    MlKemKey * key,
    int type,
    const unsigned char * id,
    int len,
    void * heap,
    int devId
)
```

Initializes an MlKemKey with a device-side key identifier. Equivalent to wc_MlKemKey_Init() but also stashes a binary id that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.

Parameters:

- **key** Pointer to the MlKemKey to initialize.
- **type** ML-KEM variant identifier.
- **id** Pointer to the device-side key identifier bytes. May be NULL when len is 0.
- **len** Number of bytes in id. Must be in the range [0, MLKEM_MAX_ID_LEN].
- **heap** Heap hint for dynamic memory allocation.
- **devId** Device identifier for the crypto callback.

See:

- wc_MlKemKey_Init
- wc_MlKemKey_Init_Label
- wc_MlKemKey_Free

Return:

- 0 on success.
- BAD_FUNC_ARG if key is NULL, if id is NULL while len is non-zero, or if type is invalid.
- BUFFER_E if len is negative or exceeds MLKEM_MAX_ID_LEN.

The id is copied into the key object; the caller may free its buffer immediately after this call returns.

```
int wc_MlKemKey_Init_Label(
    MlKemKey * key,
    int type,
    const char * label,
    void * heap,
    int devId
)
```

Initializes an MlKemKey with a device-side key label. Equivalent to wc_MlKemKey_Init() but also stashes a label string that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.

Parameters:

- **key** Pointer to the MlKemKey to initialize.
- **type** ML-KEM variant identifier.
- **label** NUL-terminated device-side key label.
- **heap** Heap hint for dynamic memory allocation.
- **devId** Device identifier for the crypto callback.

See:

- wc_MlKemKey_Init
- wc_MlKemKey_Init_Id
- wc_MlKemKey_Free

Return:

- 0 on success.
- BAD_FUNC_ARG if key or label is NULL or type is invalid.

```
int wc_MlKemKey_MakeKey(
    MlKemKey * key,
    WC_RNG * rng
)
```

Generates a new ML-KEM key pair using the provided RNG. Both the public and private components are populated on success.

Parameters:

- **key** Pointer to an initialized MlKemKey.
- **rng** Pointer to an initialized WC_RNG.

See:

- wc_MlKemKey_MakeKeyWithRandom
- wc_MlKemKey_Encapsulate
- wc_MlKemKey_Decapsulate

Return:

- 0 on success.
- BAD_FUNC_ARG if key or rng is NULL.
- MEMORY_E on allocation failure.

Example

```
MlKemKey key;
WC_RNG rng;

wc_MlKemKey_Init(&key, WC_ML_KEM_768, NULL, INVALID_DEVID);
wc_InitRng(&rng);

if (wc_MlKemKey_MakeKey(&key, &rng) != 0) {
    // error generating key pair
}
```

```
int wc_MlKemKey_MakeKeyWithRandom(
    MlKemKey * key,
    const unsigned char * rand,
    int len
)
```

Deterministic key generation: produces an ML-KEM key pair from the supplied 64 bytes of randomness instead of an RNG. Useful for known-answer tests and for applications that derive key randomness from another secret.

Parameters:

- **key** Pointer to an initialized MlKemKey.
- **rand** Pointer to a buffer of randomness.
- **len** Length of rand in bytes; must be 64.

See: wc_MlKemKey_MakeKey

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL or len is not 64.

```
int wc_MlKemKey_CipherTextSize(
    MlKemKey * key,
    word32 * len
)
```

Returns the ciphertext size in bytes for the variant selected on this key.

Parameters:

- **key** Pointer to an initialized MlKemKey.
- **len** Receives the ciphertext size in bytes.

See:

- wc_MlKemKey_SharedSecretSize
- wc_MlKemKey_Encapsulate

Return:

- 0 on success.
- BAD_FUNC_ARG if key or len is NULL.

```
int wc_MlKemKey_SharedSecretSize(
    MlKemKey * key,
    word32 * len
)
```

Returns the shared-secret size in bytes for ML-KEM. The value is the same (32 bytes) across all parameter sets but is queried programmatically for symmetry with wc_MlKemKey_CipherTextSize().

Parameters:

- **key** Pointer to an initialized MlKemKey.
- **len** Receives the shared-secret size in bytes.

See: wc_MlKemKey_CipherTextSize

Return:

- 0 on success.
- BAD_FUNC_ARG if key or len is NULL.

```
int wc_MlKemKey_Encapsulate(
    MlKemKey * key,
    unsigned char * ct,
    unsigned char * ss,
    WC_RNG * rng
)
```

Encapsulates a fresh shared secret against the public key held in key. Produces a ciphertext that the holder of the corresponding private key can pass to wc_MlKemKey_Decapsulate() to recover the same shared secret.

Parameters:

- **key** Pointer to an MlKemKey containing a public key.

- **ct** Buffer that receives the ciphertext.
- **ss** Buffer that receives the 32-byte shared secret.
- **rng** Pointer to an initialized WC_RNG.

See:

- wc_MlKemKey_EncapsulateWithRandom
- wc_MlKemKey_Decapsulate

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- BAD_STATE_E if the public key has not been set.
- NOT_COMPILED_IN if wolfSSL was built with WC_NO_RNG.
- MEMORY_E on allocation failure inside the encapsulation routine.

The ct buffer must be at least wc_MlKemKey_CipherTextSize() bytes and ss must be at least wc_MlKemKey_SharedSecretSize() bytes.

Example

```
MlKemKey key;
unsigned char ct[WC_ML_KEM_768_CIPHER_TEXT_SIZE];
unsigned char ss[WC_ML_KEM_SS_SZ];

// ... key holds the recipient's public key ...
if (wc_MlKemKey_Encapsulate(&key, ct, ss, &rng) != 0) {
    // error during encapsulation
}
// Send ct to the holder of the matching private key.
```

```
int wc_MlKemKey_EncapsulateWithRandom(
    MlKemKey * key,
    unsigned char * ct,
    unsigned char * ss,
    const unsigned char * rand,
    int len
)
```

Deterministic variant of wc_MlKemKey_Encapsulate(). Uses the supplied 32 bytes of randomness instead of consuming output from an RNG.

Parameters:

- **key** Pointer to an MlKemKey containing a public key.
- **ct** Buffer that receives the ciphertext.
- **ss** Buffer that receives the 32-byte shared secret.
- **rand** Buffer of randomness.
- **len** Length of rand in bytes; must be 32.

See: wc_MlKemKey_Encapsulate

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL or len is not 32.

```
int wc_MlKemKey_Decapsulate(
    MlKemKey * key,
    unsigned char * ss,
    const unsigned char * ct,
    word32 len
)
```

Decapsulates a ciphertext using the private key held in key and recovers the shared secret produced by wc_MlKemKey_Encapsulate(). ML_KEM decapsulation is constant time and includes an implicit_rejection check on malformed ciphertexts (an attacker cannot learn the validity of ct from the runtime).

Parameters:

- **key** Pointer to an MlKemKey with the private key.
- **ss** Buffer that receives the 32-byte shared secret.
- **ct** The ciphertext to decapsulate.
- **len** Length of ct in bytes.

See:

- wc_MlKemKey_Encapsulate
- wc_MlKemKey_CipherTextSize

Return:

- 0 on success (a shared secret was written to ss).
- BAD_FUNC_ARG if any required pointer is NULL.
- BAD_STATE_E if the private key has not been set.
- BUFFER_E if len does not match the expected ciphertext size for the configured ML-KEM variant.
- NOT_COMPILED_IN if the key's ML-KEM variant was disabled at build time.
- MEMORY_E on allocation failure.

The ss buffer must be at least wc_MlKemKey_SharedSecretSize() bytes and ct must be exactly wc_MlKemKey_CipherTextSize() bytes.

```
int wc_MlKemKey_DecodePrivateKey(
    MlKemKey * key,
    const unsigned char * in,
    word32 len
)
```

Decodes a raw ML-KEM private key into key. The variant must already be selected on the key (typically via wc_MlKemKey_Init() or wc_MlKemKey_New()) and len must match the private key size for that variant.

Parameters:

- **key** Pointer to an initialized MlKemKey.
- **in** Raw private key bytes.
- **len** Length of in in bytes.

See:

- wc_MlKemKey_EncodePrivateKey
- wc_MlKemKey_PrivateKeySize

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL or len does not match the expected size.

```
int wc_MlKemKey_DecodePublicKey(
    MlKemKey * key,
    const unsigned char * in,
    word32 len
)
```

Decodes a raw ML-KEM public key into key. The variant must already be selected on the key and len must match the public key size for that variant.

Parameters:

- **key** Pointer to an initialized MlKemKey.
- **in** Raw public key bytes.
- **len** Length of in in bytes.

See:

- wc_MlKemKey_EncodePublicKey
- wc_MlKemKey_PublicKeySize

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL or len does not match the expected size.

```
int wc_MlKemKey_PrivateKeySize(
    MlKemKey * key,
    word32 * len
)
```

Returns the encoded private key size in bytes for the variant selected on this key.

Parameters:

- **key** Pointer to an initialized MlKemKey.
- **len** Receives the private key size in bytes.

See: wc_MlKemKey_PublicKeySize

Return:

- 0 on success.
- BAD_FUNC_ARG if key or len is NULL.

```
int wc_MlKemKey_PublicKeySize(
    MlKemKey * key,
    word32 * len
)
```

Returns the encoded public key size in bytes for the variant selected on this key.

Parameters:

- **key** Pointer to an initialized MlKemKey.
- **len** Receives the public key size in bytes.

See: wc_MlKemKey_PrivateKeySize

Return:

- 0 on success.
- BAD_FUNC_ARG if key or len is NULL.

```
int wc_MlKemKey_EncodePrivateKey(
    MlKemKey * key,
    unsigned char * out,
    word32 len
)
```

Encodes the ML-KEM private key into out. The out buffer length must be exactly wc_MlKemKey_PrivateKeySize() bytes.

Parameters:

- **key** Pointer to an MlKemKey with a private key.
- **out** Buffer that receives the encoded private key.
- **len** Length of out in bytes.

See:

- wc_MlKemKey_DecodePrivateKey
- wc_MlKemKey_PrivateKeySize

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- BAD_STATE_E if the private and public keys are not both set on the key object.
- BUFFER_E if len does not exactly equal the encoded private key size for the configured ML-KEM variant.
- NOT_COMPILED_IN if the key's ML-KEM variant was disabled at build time.

```
int wc_MlKemKey_EncodePublicKey(
    MlKemKey * key,
    unsigned char * out,
    word32 len
)
```

Encodes the ML-KEM public key into out. The out buffer length must be exactly wc_MlKemKey_PublicKeySize() bytes.

Parameters:

- **key** Pointer to an MlKemKey with a public key.
- **out** Buffer that receives the encoded public key.
- **len** Length of out in bytes.

See:

- wc_MlKemKey_DecodePublicKey
- wc_MlKemKey_PublicKeySize

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- BAD_STATE_E if the public key has not been set.
- BUFFER_E if len does not exactly equal the encoded public key size for the configured ML-KEM variant.
- NOT_COMPILED_IN if the key's ML-KEM variant was disabled at build time.

```
int wc_MlKemKey_PublicKeyToDer(
    MlKemKey * key,
    byte * output,
    word32 len,
    int withAlg
)
```

Encodes the ML-KEM public key as a DER SubjectPublicKeyInfo. Pass NULL for output to obtain the required length.

Parameters:

- **key** Pointer to an MlKemKey with a public key.
- **output** Buffer that receives the DER, or NULL for a length.
- **len** Length of output in bytes.
- **withAlg** Include the SubjectPublicKeyInfo wrapper when 1, or emit only the raw public key bytes when 0.

See:

- wc_MlKemKey_PublicKeyDecode
- wc_MlKemKey_PrivateKeyToDer

Return:

- Length of the encoding in bytes on success.
- BAD_FUNC_ARG if key is NULL or its type has no standardised OID.
- MEMORY_E if dynamic memory allocation fails.

```
int wc_MlKemKey_PrivateKeyToDer(
    MlKemKey * key,
    byte * output,
    word32 len
)
```

Encodes the ML-KEM private key as a DER PKCS#8 OneAsymmetricKey, carrying the expanded decapsulation key. Pass NULL for output to obtain the required length.

Parameters:

- **key** Pointer to an MlKemKey with a private key.
- **output** Buffer that receives the DER, or NULL for a length.
- **len** Length of output in bytes.

See:

- wc_MlKemKey_PrivateKeyDecode
- wc_MlKemKey_PublicKeyToDer

Return:

- Length of the encoding in bytes on success.
- BAD_FUNC_ARG if key is NULL or its type has no standardised OID.
- MEMORY_E if dynamic memory allocation fails.

```
int wc_MlKemKey_PublicKeyDecode(
    MlKemKey * key,
    const byte * input,
    word32 inSz,
    word32 * inOutIdx
)
```

Decodes a DER SubjectPublicKeyInfo into an ML-KEM public key. Takes wrapped DER, where wc_MlKemKey_DecodePublicKey takes the raw encoded key. A key initialized for a parameter set holds the DER to that same one. A key initialized with WC_ML_KEM_TYPE_UNSET takes the parameter set from the algorithm OID in the DER instead.

Parameters:

- **key** Pointer to an initialized MlKemKey.
- **input** Buffer holding the DER.
- **inSz** Length of input in bytes.
- **inOutIdx** On in, index into input; on out, index after.

See:

- wc_MlKemKey_PublicKeyToDer
- wc_MlKemKey_Init

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- ASN_PARSE_E if the DER is invalid or names another parameter set.
- NOT_COMPILED_IN if the DER names a parameter set this build lacks.

```
int wc_MlKemKey_PrivateKeyDecode(
    MlKemKey * key,
    const byte * input,
    word32 inSz,
    word32 * inOutIdx
)
```

Decodes a DER PKCS#8 OneAsymmetricKey into an ML-KEM private key. Takes wrapped DER, where wc_MlKemKey_DecodePrivateKey takes the raw encoded key. All three RFC 9935 Section 6 CHOICE forms are accepted: the 64-byte seed under an implicit [0], the expanded decapsulation key as an OCTET STRING, and the SEQUENCE carrying both. A seed is expanded with

ML-KEM.KeyGen_internal(d,z). For the “both” form the expanded key is regenerated from the seed and the two are compared, so a key whose halves disagree is rejected as malformed, as RFC 9935 Section 8 requires. A key initialized for a parameter set holds the DER to that same parameter set; a key initialized with WC_ML_KEM_TYPE_UNSET takes it from the algorithm OID in the DER instead.

Parameters:

- **key** Pointer to an initialized MlKemKey.
- **input** Buffer holding the DER.
- **inSz** Length of input in bytes.
- **inOutIdx** On in, index into input; on out, index after.

See: wc_MlKemKey_PrivateKeyToDer

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- MEMORY_E if dynamic memory allocation fails.
- ASN_PARSE_E if the DER is invalid, names another parameter set, carries a seed of the wrong length, or pairs a seed with an expanded key that does not match it.
- NOT_COMPILED_IN if the key carries a seed and the build defines WOLFSSL_MLKEM_NO_MAKE_KEY or WOLF_CRYPTO_CB_ONLY_MLKEM.

A build defining WOLFSSL_MLKEM_NO_MAKE_KEY or WOLF_CRYPTO_CB_ONLY_MLKEM cannot expand a seed, and so cannot perform the Section 8 comparison either. Such a build rejects every key that carries a seed, the “both” form included, rather than accepting its expanded half unchecked.

B.41 Algorithms - Password Based

B.40.3.23 function wc_MlKemKey_PrivateKeyDecode

B.41.1 Functions

	Name
int	**wc_PBKDF1**(byte * output, const byte * passwd, int pLen, const byte * salt, int sLen, int iterations, int kLen, int hashType)This function implements the Password Based Key Derivation Function 1 (PBKDF1), converting an input password with a concatenated salt into a more secure key, which it stores in output. It allows the user to select between SHA and MD5 as hash functions.

	Name
int	**wc_PBKDF2**(byte * output, const byte * passwd, int pLen, const byte * salt, int sLen, int iterations, int kLen, int hashType)This function implements the Password Based Key Derivation Function 2 (PBKDF2), converting an input password with a concatenated salt into a more secure key, which it stores in output. It allows the user to select any of the supported HMAC hash functions, including: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512.
int	**wc_PKCS12_PBKDF**(byte * output, const byte * passwd, int passLen, const byte * salt, int saltLen, int iterations, int kLen, int hashType, int id)This function implements the Password Based Key Derivation Function (PBKDF) described in RFC 7292 Appendix B. This function converts an input password with a concatenated salt into a more secure key, which it stores in output. It allows the user to select any of the supported HMAC hash functions, including: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512.
int	**wc_PBKDF1_ex**(byte * key, int keyLen, byte * iv, int ivLen, const byte * passwd, int passwdLen, const byte * salt, int saltLen, int iterations, int hashType, void * heap)Extended version of PBKDF1 with heap hint.
int	**wc_PBKDF2_ex**(byte * output, const byte * passwd, int pLen, const byte * salt, int sLen, int iterations, int kLen, int hashType, void * heap, int devId)Extended version of PBKDF2 with heap hint and device ID.
int	**wc_PKCS12_PBKDF_ex**(byte * output, const byte * passwd, int passLen, const byte * salt, int saltLen, int iterations, int kLen, int hashType, int id, void * heap)Extended version of PKCS12_PBKDF with heap hint.
int	**wc_scrypt**(byte * output, const byte * passwd, int passLen, const byte * salt, int saltLen, int cost, int blockSize, int parallel, int dkLen)Implements scrypt key derivation function.
int	**wc_scrypt_ex**(byte * output, const byte * passwd, int passLen, const byte * salt, int saltLen, word32 iterations, int blockSize, int parallel, int dkLen)Extended scrypt with iteration count instead of cost.

	Name
int	**wc_PBKDF_max_iterations_set**(int iters)Set the current iteration limit for PBKDF.
int	**wc_PBKDF_max_iterations_get**(void)Get the current iteration limit for PBKDF.

B.41.2 Functions Documentation

```
int wc_PBKDF1(
    byte * output,
    const byte * passwd,
    int pLen,
    const byte * salt,
    int sLen,
    int iterations,
    int kLen,
    int hashType
)
```

This function implements the Password Based Key Derivation Function 1 (PBKDF1), converting an input password with a concatenated salt into a more secure key, which it stores in output. It allows the user to select between SHA and MD5 as hash functions.

Parameters:

- **output** pointer to the buffer in which to store the generated key. Should be at least kLen long
- **passwd** pointer to the buffer containing the password to use for the key derivation
- **pLen** length of the password to use for key derivation
- **salt** pointer to the buffer containing the salt to use for key derivation
- **sLen** length of the salt
- **iterations** number of times to process the hash
- **kLen** desired length of the derived key. Should not be longer than the digest size of the hash chosen
- **hashType** the hashing algorithm to use. Valid choices are WC_MD5 and WC_SHA

See:

- wc_PBKDF2
- wc_PKCS12_PBKDF

Return:

- 0 Returned on successfully deriving a key from the input password
- BAD_FUNC_ARG Returned if there is an invalid hash type given (valid type are: MD5 and SHA), iterations is less than 1, or the key length (kLen) requested is greater than the hash length of the provided hash
- MEMORY_E Returned if there is an error allocating memory for a SHA or MD5 object

Example

```
int ret;
byte key[WC_MD5_DIGEST_SIZE];
byte pass[] = { }; // initialize with password
byte salt[] = { }; // initialize with salt

ret = wc_PBKDF1(key, pass, sizeof(pass), salt, sizeof(salt), 1000,
sizeof(key), WC_MD5);
if ( ret != 0 ) {
    // error deriving key from password
}
```

```
int wc_PBKDF2(
    byte * output,
    const byte * passwd,
    int pLen,
    const byte * salt,
    int sLen,
    int iterations,
    int kLen,
    int hashType
)
```

This function implements the Password Based Key Derivation Function 2 (PBKDF2), converting an input password with a concatenated salt into a more secure key, which it stores in output. It allows the user to select any of the supported HMAC hash functions, including: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512.

Parameters:

- **output** pointer to the buffer in which to store the generated key. Should be kLen long
- **passwd** pointer to the buffer containing the password to use for the key derivation
- **pLen** length of the password to use for key derivation
- **salt** pointer to the buffer containing the salt to use for key derivation
- **sLen** length of the salt
- **iterations** number of times to process the hash
- **kLen** desired length of the derived key
- **hashType** the hashing algorithm to use. Valid choices are: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512

See:

- wc_PBKDF1
- wc_PKCS12_PBKDF

Return:

- 0 Returned on successfully deriving a key from the input password
- BAD_FUNC_ARG Returned if there is an invalid hash type given or iterations is less than 1
- MEMORY_E Returned if there is an allocating memory for the HMAC object

Example

```
int ret;
byte key[64];
byte pass[] = { }; // initialize with password
byte salt[] = { }; // initialize with salt

ret = wc_PBKDF2(key, pass, sizeof(pass), salt, sizeof(salt), 2048, sizeof(key),
WC_SHA512);
if ( ret != 0 ) {
    // error deriving key from password
}
```

```
int wc_PKCS12_PBKDF(
    byte * output,
    const byte * passwd,
    int passLen,
    const byte * salt,
    int saltLen,
    int iterations,
    int kLen,
    int hashType,
    int id
)
```

This function implements the Password Based Key Derivation Function (PBKDF) described in RFC 7292 Appendix B. This function converts an input password with a concatenated salt into a more secure key, which it stores in output. It allows the user to select any of the supported HMAC hash functions, including: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512.

Parameters:

- **output** pointer to the buffer in which to store the generated key. Should be kLen long
- **passwd** pointer to the buffer containing the password to use for the key derivation
- **passLen** length of the password to use for key derivation
- **salt** pointer to the buffer containing the salt to use for key derivation
- **saltLen** length of the salt
- **iterations** number of times to process the hash
- **kLen** desired length of the derived key
- **hashType** the hashing algorithm to use. Valid choices are: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512
- **id** this is a byte identifier indicating the purpose of key generation. It is used to diversify the key output, and should be assigned as follows: ID=1: pseudorandom bits are to be used as key material for performing encryption or decryption. ID=2: pseudorandom bits are to be used an IV (Initial Value) for encryption or decryption. ID=3: pseudorandom bits are to be used as an integrity key for MACing.

See:

- wc_PBKDF1
- wc_PBKDF2

Return:

- 0 Returned on successfully deriving a key from the input password
- BAD_FUNC_ARG Returned if there is an invalid hash type given, iterations is less than 1, or the key length (kLen) requested is greater than the hash length of the provided hash
- MEMORY_E Returned if there is an allocating memory
- MP_INIT_E may be returned if there is an error during key generation
- MP_READ_E may be returned if there is an error during key generation
- MP_CMP_E may be returned if there is an error during key generation
- MP_INVMOD_E may be returned if there is an error during key generation
- MP_EXPTMOD_E may be returned if there is an error during key generation
- MP_MOD_E may be returned if there is an error during key generation
- MP_MUL_E may be returned if there is an error during key generation
- MP_ADD_E may be returned if there is an error during key generation
- MP_MULMOD_E may be returned if there is an error during key generation
- MP_TO_E may be returned if there is an error during key generation
- MP_MEM may be returned if there is an error during key generation

Example

```
int ret;
byte key[64];
byte pass[] = { }; // initialize with password
byte salt[] = { }; // initialize with salt

ret = wc_PKCS12_PBKDF(key, pass, sizeof(pass), salt, sizeof(salt), 2048,
sizeof(key), WC_SHA512, 1);
if ( ret != 0 ) {
    // error deriving key from password
}
```

```
int wc_PBKDF1_ex(
    byte * key,
    int keyLen,
    byte * iv,
    int ivLen,
    const byte * passwd,
    int passwdLen,
    const byte * salt,
    int saltLen,
    int iterations,
    int hashType,
    void * heap
)
```

Extended version of PBKDF1 with heap hint.

Parameters:

- **key** Output key buffer

- **keyLen** Key length
- **iv** Output IV buffer
- **ivLen** IV length
- **passwd** Password buffer
- **passwdLen** Password length
- **salt** Salt buffer
- **saltLen** Salt length
- **iterations** Iteration count
- **hashType** Hash algorithm type
- **heap** Heap hint for memory allocation

See:

- wc_PBKDF1
- wc_PBKDF_max_iterations_set
- wc_PBKDF_max_iterations_get

Return:

- 0 on success
- BAD_FUNC_ARG on invalid arguments or iterations is greater than current_wc_pbkdf_max_iterations
- MEMORY_E on memory allocation error

Example

```
byte key[16], iv[16];
byte pass[] = "password";
byte salt[] = "salt";
int ret = wc_PBKDF1_ex(key, sizeof(key), iv, sizeof(iv),
    pass, sizeof(pass), salt, sizeof(salt), 1000, WC_SHA, NULL);
```

```
int wc_PBKDF2_ex(
    byte * output,
    const byte * passwd,
    int pLen,
    const byte * salt,
    int sLen,
    int iterations,
    int kLen,
    int hashType,
    void * heap,
    int devId
)
```

Extended version of PBKDF2 with heap hint and device ID.

Parameters:

- **output** Output key buffer
- **passwd** Password buffer

- **pLen** Password length
- **salt** Salt buffer
- **sLen** Salt length
- **iterations** Iteration count
- **kLen** Key length
- **hashType** Hash algorithm type
- **heap** Heap hint for memory allocation
- **devId** Device ID for hardware acceleration

See:

- wc_PBKDF2
- wc_PBKDF_max_iterations_set
- wc_PBKDF_max_iterations_get

Return:

- 0 on success
- BAD_FUNC_ARG on invalid arguments or iterations is greater than current_wc_pbkdf_max_iterations
- MEMORY_E on memory allocation error

Example

```
byte key[32];
byte pass[] = "password";
byte salt[] = "salt";
int ret = wc_PBKDF2_ex(key, pass, sizeof(pass), salt,
    sizeof(salt), 2048, sizeof(key), WC_SHA256, NULL,
    INVALID_DEVID);
```

```
int wc_PKCS12_PBKDF_ex(
    byte * output,
    const byte * passwd,
    int passLen,
    const byte * salt,
    int saltLen,
    int iterations,
    int kLen,
    int hashType,
    int id,
    void * heap
)
```

Extended version of PKCS12_PBKDF with heap hint.

Parameters:

- **output** Output key buffer
- **passwd** Password buffer
- **passLen** Password length

- **salt** Salt buffer
- **saltLen** Salt length
- **iterations** Iteration count
- **kLen** Key length
- **hashType** Hash algorithm type
- **id** Purpose identifier (1=key, 2=IV, 3=MAC)
- **heap** Heap hint for memory allocation

See:

- wc_PKCS12_PBKDF
- wc_PBKDF_max_iterations_set
- wc_PBKDF_max_iterations_get

Return:

- 0 on success
- BAD_FUNC_ARG on invalid arguments or iterations is greater than current_wc_pbkdf_max_iterations
- MEMORY_E on memory allocation error

Example

```
byte key[32];
byte pass[] = "password";
byte salt[] = "salt";
int ret = wc_PKCS12_PBKDF_ex(key, pass, sizeof(pass), salt,
    sizeof(salt), 2048, sizeof(key), WC_SHA256, 1, NULL);
```

```
int wc_scrypt(
    byte * output,
    const byte * passwd,
    int passLen,
    const byte * salt,
    int saltLen,
    int cost,
    int blockSize,
    int parallel,
    int dkLen
)
```

Implements scrypt key derivation function.

Parameters:

- **output** Output key buffer
- **passwd** Password buffer
- **passLen** Password length
- **salt** Salt buffer
- **saltLen** Salt length
- **cost** CPU/memory cost parameter (N)

- **blockSize** Block size parameter (r)
- **parallel** Parallelization parameter (p)
- **dkLen** Derived key length

See: wc_scrypt_ex

Return:

- 0 on success
- BAD_FUNC_ARG on invalid arguments
- MEMORY_E on memory allocation error

Example

```
byte key[32];
byte pass[] = "password";
byte salt[] = "salt";
int ret = wc_scrypt(key, pass, sizeof(pass), salt,
    sizeof(salt), 16384, 8, 1, sizeof(key));
```

```
int wc_scrypt_ex(
    byte * output,
    const byte * passwd,
    int passLen,
    const byte * salt,
    int saltLen,
    word32 iterations,
    int blockSize,
    int parallel,
    int dkLen
)
```

Extended scrypt with iteration count instead of cost.

Parameters:

- **output** Output key buffer
- **passwd** Password buffer
- **passLen** Password length
- **salt** Salt buffer
- **saltLen** Salt length
- **iterations** Iteration count
- **blockSize** Block size parameter (r)
- **parallel** Parallelization parameter (p)
- **dkLen** Derived key length

See: wc_scrypt

Return:

- 0 on success

- BAD_FUNC_ARG on invalid arguments
- MEMORY_E on memory allocation error

Example

```
byte key[32];
byte pass[] = "password";
byte salt[] = "salt";
int ret = wc_scrypt_ex(key, pass, sizeof(pass), salt,
    sizeof(salt), 16384, 8, 1, sizeof(key));
```

```
int wc_PBKDF_max_iterations_set(
    int iters
)
```

Set the current iteration limit for PBKDF.

Parameters:

- **iters** The new iteration limit.

See: wc_scrypt

Return:

- Previous iteration limit on success
- BAD_FUNC_ARG on invalid arguments

By default, the iteration limit is set to WC_PBKDF_DEFAULT_MAX_ITERATIONS, which can be overridden at build time. This function allows runtime override of the limit.

Note that [wc_PBKDF_max_iterations_set()](#function-wc_pbkdf_max_iterations_set) has no provisions for thread synchronization. Users should arrange to call it at startup or idle times, when there are no other PBKDF calls in progress.

Example

```
int prev_iter_limit = wc_PBKDF_max_iterations_set(100000000);
```

```
int wc_PBKDF_max_iterations_get(
    void
)
```

Get the current iteration limit for PBKDF.

See: wc_scrypt

Return: Current iteration limit

Example

```
int cur_iter_limit = wc_PBKDF_max_iterations_get();
```

B.42 Algorithms - PKCS7

B.41.2.10 function wc_PBKDF_max_iterations_get

B.42.1 Types

	Name
typedef int()(*const byte* key, word32 keySz, const byte *in, word32 inSz, int wrap, byte* out, word32 outSz)	**CallbackAESKeyWrapUnwrap** Callback used for a custom AES key wrap/unwrap operation.

B.42.2 Functions

	Name
int	**wc_PKCS7_InitWithCert**(wc_PKCS7 * pkcs7, byte * der, word32 derSz)This function initializes a PKCS7 structure with a DER-formatted certificate. To initialize an empty PKCS7 structure, one can pass in a NULL cert and 0 for certSz.
void	**wc_PKCS7_Free**(wc_PKCS7 * pkcs7)This function releases any memory allocated by a PKCS7 initializer.
int	**wc_PKCS7_EncodeData**(wc_PKCS7 * pkcs7, byte * output, word32 outputSz)This function builds the PKCS7 data content type, encoding the PKCS7 structure into a buffer containing a parsable PKCS7 data packet.
int	**wc_PKCS7_EncodeSignedData**(wc_PKCS7 * pkcs7, byte * output, word32 outputSz)This function builds the PKCS7 signed data content type, encoding the PKCS7 structure into a buffer containing a parsable PKCS7 signed data packet. For RSA_PSS signers (WC_RSA_PSS), see PKCS#7 RSA_PSS (CMS).
int	**wc_PKCS7_EncodeSignedData_ex**(wc_PKCS7 * pkcs7, const byte * hashBuf, word32 hashSz, byte * outputHead, word32 * outputHeadSz, byte * outputFoot, word32 * outputFootSz)This function builds the PKCS7 signed data content type, encoding the PKCS7 structure into a header and footer buffer containing a parsable PKCS7 signed data packet. This does not include the content. A hash must be computed and provided for the data.

	Name
int	**wc_PKCS7_VerifySignedData**(wc_PKCS7 * pkcs7, byte * pkiMsg, word32 pkiMsgSz)This function takes in a transmitted PKCS7 signed data message, extracts the certificate list and certificate revocation list, and then verifies the signature. It stores the extracted content in the given PKCS7 structure.
int	**wc_PKCS7_VerifySignedData_ex**(wc_PKCS7 * pkcs7, const byte * hashBuf, word32 hashSz, byte * pkiMsgHead, word32 pkiMsgHeadSz, byte * pkiMsgFoot, word32 pkiMsgFootSz)This function takes in a transmitted PKCS7 signed data message as hash/header/footer, then extracts the certificate list and certificate revocation list, and then verifies the signature. It stores the extracted content in the given PKCS7 structure.
int	**wc_PKCS7_SetAESKeyWrapUnwrapCb**(wc_PKCS7 * pkcs7, CallbackAESKeyWrapUnwrap aesKeyWrapCb)Set the callback function to be used to perform a custom AES key wrap/unwrap operation.
int	**wc_PKCS7_EncodeEnvelopedData**(wc_PKCS7 * pkcs7, byte * output, word32 outputSz)This function builds the PKCS7 enveloped data content type, encoding the PKCS7 structure into a buffer containing a parsable PKCS7 enveloped data packet.
int	**wc_PKCS7_DecodeEnvelopedData**(wc_PKCS7 * pkcs7, byte * pkiMsg, word32 pkiMsgSz, byte * output, word32 outputSz)This function unwraps and decrypts a PKCS7 enveloped data content type, decoding the message into output. It uses the private key of the PKCS7 object passed in to decrypt the message.
int	**wc_PKCS7_GetEnvelopedDataKariRid**(const byte * in, word32 inSz, byte * out, word32 * outSz)This function extracts the KeyAgreeRecipientIdentifier object from an EnvelopedData package containing a KeyAgreeRecipientInfo RecipientInfo object. Only the first KeyAgreeRecipientIdentifer found in the first RecipientInfo is copied. This function does not support multiple RecipientInfo objects or multiple RecipientEncryptedKey objects within an KeyAgreeRecipientInfo.

	Name
int	**wc_PKCS7_DecodeEncryptedData**(wc_PKCS7 * pkcs7, byte * pkiMsg, word32 pkiMsgSz, byte * output, word32 outputSz)This function unwraps and decrypts a PKCS7 encrypted data content type, decoding the message into output. It uses the encryption key of the PKCS7 object passed in via pkcs7->encryptionKey and pkcs7->encryptionKeySz to decrypt the message.
int	**wc_PKCS7_DecodeEncryptedKeyPackage**(wc_PKCS7 * pkcs7, byte * pkiMsg, word32 pkiMsgSz, byte * output, word32 outputSz)This function unwraps and decrypts a PKCS7 encrypted key package content type, decoding the message into output. If the wrapped content type is EncryptedData, the encryption key must be set in the pkcs7 input structure (via pkcs7_>encryptionKey and pkcs7_>encryptionKeySz). If the wrapped content type is EnvelopedData, the private key must be set in the pkcs7 input structure (via pkcs7_>privateKey and pkcs7_>privateKeySz). A wrapped content type of AuthEnvelopedData is not currently supported.
int	**wc_PKCS7_DecodeSymmetricKeyPackageAttribute**(const byte * skp, word32 skpSz, size_t index, const byte ** attr, word32 * attrSz)This function provides access to a SymmetricKeyPackage attribute.
int	**wc_PKCS7_DecodeSymmetricKeyPackageKey**(const byte * skp, word32 skpSz, size_t index, const byte ** key, word32 * keySz)This function provides access to a SymmetricKeyPackage key.
int	**wc_PKCS7_DecodeOneSymmetricKeyAttribute**(const byte * osk, word32 oskSz, size_t index, const byte ** attr, word32 * attrSz)This function provides access to a OneSymmetricKey attribute.
int	**wc_PKCS7_DecodeOneSymmetricKeyKey**(const byte * osk, word32 oskSz, const byte ** key, word32 * keySz)This function provides access to a OneSymmetricKey key.
PKCS7 *	**wolfSSL_PKCS7_new**(void)Creates new PKCS7 structure.
PKCS7_SIGNED *	**wolfSSL_PKCS7_SIGNED_new**(void)Creates new PKCS7_SIGNED structure.
void	**wolfSSL_PKCS7_free**(PKCS7 * p7)Frees PKCS7 structure.
void	**wolfSSL_PKCS7_SIGNED_free**(PKCS7_SIGNED * p7)Frees PKCS7_SIGNED structure.

	Name
PKCS7 *	**wolfSSL_d2i_PKCS7**(PKCS7 ** p7, const unsigned char ** in, int len)Decodes DER-encoded PKCS7 structure.
PKCS7 *	**wolfSSL_d2i_PKCS7_bio**(WOLFSSL_BIO * bio, PKCS7 ** p7)Decodes PKCS7 from BIO.
int	**wolfSSL_i2d_PKCS7_bio**(WOLFSSL_BIO * bio, PKCS7 * p7)Encodes PKCS7 to BIO.
int	**wolfSSL_i2d_PKCS7**(PKCS7 * p7, unsigned char ** out)Encodes PKCS7 to DER.
PKCS7 *	**wolfSSL_PKCS7_sign**(WOLFSSL_X509 * signer, WOLFSSL_EVP_PKEY * pkey, WOLFSSL_STACK * certs, WOLFSSL_BIO * in, int flags)Creates signed PKCS7 message.
int	**wolfSSL_PKCS7_verify**(PKCS7 * p7, WOLFSSL_STACK * certs, WOLFSSL_X509_STORE * store, WOLFSSL_BIO * in, WOLFSSL_BIO * out, int flags)Verifies signed PKCS7 message.
int	**wolfSSL_PKCS7_final**(PKCS7 * pkcs7, WOLFSSL_BIO * in, int flags)Finalizes PKCS7 structure with data.
int	**wolfSSL_PKCS7_encode_certs**(PKCS7 * p7, WOLFSSL_STACK * certs, WOLFSSL_BIO * out)Encodes certificates into PKCS7.
WOLFSSL_STACK *	**wolfSSL_PKCS7_to_stack**(PKCS7 * pkcs7)Converts PKCS7 certificates to stack.
WOLFSSL_STACK *	**wolfSSL_PKCS7_get0_signers**(PKCS7 * p7, WOLFSSL_STACK * certs, int flags)Gets signer certificates from PKCS7.
int	**wolfSSL_PEM_write_bio_PKCS7**(WOLFSSL_BIO * bio, PKCS7 * p7)Writes PKCS7 to BIO in PEM format.
PKCS7 *	**wolfSSL_SMIME_read_PKCS7**(WOLFSSL_BIO * in, WOLFSSL_BIO ** bcont)Reads S/MIME PKCS7 from BIO.
int	**wolfSSL_SMIME_write_PKCS7**(WOLFSSL_BIO * out, PKCS7 * pkcs7, WOLFSSL_BIO * in, int flags)Writes PKCS7 to BIO in S/MIME format.
wc_PKCS7 *	**wc_PKCS7_New**(void * heap, int devId)Creates new wc_PKCS7 structure.
void	**wc_PKCS7_SetUnknownExtCallback**(wc_PKCS7 * pkcs7, wc_UnknownExtCallback cb)Sets unknown extension callback.
int	**wc_PKCS7_Init**(wc_PKCS7 * pkcs7, void * heap, int devId)Initializes wc_PKCS7 structure.
int	**wc_PKCS7_AddCertificate**(wc_PKCS7 * pkcs7, byte * der, word32 derSz)Adds certificate to PKCS7.
int	**wc_PKCS7_GetAttributeValue**(wc_PKCS7 * pkcs7, const byte * oid, word32 oidSz, byte * out, word32 * outSz)Gets attribute value from PKCS7.

	Name
int	**wc_PKCS7_SetSignerIdentifierType**(wc_PKCS7 * pkcs7, int type)Sets signer identifier type.
int	**wc_PKCS7_SetContentType**(wc_PKCS7 * pkcs7, byte * contentType, word32 sz)Sets content type.
int	**wc_PKCS7_GetPadSize**(word32 inputSz, word32 blockSz)Gets padding size for block cipher.
int	**wc_PKCS7_PadData**(byte * in, word32 inSz, byte * out, word32 outSz, word32 blockSz)Pads data for block cipher.
int	**wc_PKCS7_SetCustomSKID**(wc_PKCS7 * pkcs7, const byte * in, word16 inSz)Sets custom subject key identifier.
int	**wc_PKCS7_SetDetached**(wc_PKCS7 * pkcs7, word16 flag)Sets detached signature flag.
int	**wc_PKCS7_NoDefaultSignedAttribs**(wc_PKCS7 * pkcs7)Disables default signed attributes.
int	**wc_PKCS7_SetDefaultSignedAttribs**(wc_PKCS7 * pkcs7, word16 flag)Sets default signed attributes flag.
void	**wc_PKCS7_AllowDegenerate**(wc_PKCS7 * pkcs7, word16 flag)Allows degenerate PKCS7 (no signers).
int	**wc_PKCS7_GetSignerSID**(wc_PKCS7 * pkcs7, byte * out, word32 * outSz)Gets signer subject identifier.
int	**wc_PKCS7_EncodeSignedFPD**(wc_PKCS7 * pkcs7, byte * privateKey, word32 privateKeySz, int signOID, int hashOID, byte * content, word32 contentSz, PKCS7Attrib * signedAttribs, word32 signedAttribsSz, byte * output, word32 outputSz)Encodes signed FirmwarePackageData.
int	**wc_PKCS7_EncodeSignedEncryptedFPD**(wc_PKCS7 * pkcs7, byte * encryptKey, word32 encryptKeySz, byte * privateKey, word32 privateKeySz, int encryptOID, int signOID, int hashOID, byte * content, word32 contentSz, PKCS7Attrib * unprotectedAttribs, word32 unprotectedAttribsSz, PKCS7Attrib * signedAttribs, word32 signedAttribsSz, byte * output, word32 outputSz)Encodes signed encrypted FirmwarePackageData.
int	**wc_PKCS7_EncodeSignedCompressedFPD**(wc_PKCS7 * pkcs7, byte * privateKey, word32 privateKeySz, int signOID, int hashOID, byte * content, word32 contentSz, PKCS7Attrib * signedAttribs, word32 signedAttribsSz, byte * output, word32 outputSz)Encodes signed compressed FirmwarePackageData.

	Name
int	**wc_PKCS7_EncodeSignedEncryptedCompressedFPD**(wc_PKCS7 * pkcs7, byte * encryptKey, word32 encryptKeySz, byte * privateKey, word32 privateKeySz, int encryptOID, int signOID, int hashOID, byte * content, word32 contentSz, PKCS7Attrib * unprotectedAttribs, word32 unprotectedAttribsSz, PKCS7Attrib * signedAttribs, word32 signedAttribsSz, byte * output, word32 outputSz)Encodes signed encrypted compressed FirmwarePackageData.
int	**wc_PKCS7_AddRecipient_KTRI**(wc_PKCS7 * pkcs7, const byte * cert, word32 certSz, int options)Adds KTRI recipient.
int	**wc_PKCS7_AddRecipient_KARI**(wc_PKCS7 * pkcs7, const byte * cert, word32 certSz, int keyWrapOID, int keyAgreeOID, byte * ukm, word32 ukmSz, int options)Adds KARI recipient.
int	**wc_PKCS7_SetKey**(wc_PKCS7 * pkcs7, byte * key, word32 keySz)Sets encryption key.
int	**wc_PKCS7_AddRecipient_KEKRI**(wc_PKCS7 * pkcs7, int keyWrapOID, byte * kek, word32 kekSz, byte * keyID, word32 keyIdSz, void * timePtr, byte * otherOID, word32 otherOIDSz, byte * other, word32 otherSz, int options)Adds KEKRI recipient.
int	**wc_PKCS7_SetPassword**(wc_PKCS7 * pkcs7, byte * passwd, word32 pLen)Sets password for PWRI.
int	**wc_PKCS7_AddRecipient_PWRI**(wc_PKCS7 * pkcs7, byte * passwd, word32 pLen, byte * salt, word32 saltSz, int kdfOID, int prfOID, int iterations, int kekEncryptOID, int options)Adds PWRI recipient.
int	**wc_PKCS7_SetOriEncryptCtx**(wc_PKCS7 * pkcs7, void * ctx)Sets originator encryption context.
int	**wc_PKCS7_SetOriDecryptCtx**(wc_PKCS7 * pkcs7, void * ctx)Sets originator decryption context.
int	**wc_PKCS7_SetOriDecryptCb**(wc_PKCS7 * pkcs7, CallbackOriDecrypt cb)Sets originator decryption callback.
int	**wc_PKCS7_AddRecipient_ORI**(wc_PKCS7 * pkcs7, CallbackOriEncrypt cb, int options)Adds ORI recipient.
int	**wc_PKCS7_SetWrapCEKCb**(wc_PKCS7 * pkcs7, CallbackWrapCEK wrapCEKCb)Sets CEK wrap callback.
int	**wc_PKCS7_SetRsaSignRawDigestCb**(wc_PKCS7 * pkcs7, CallbackRsaSignRawDigest cb)Sets RSA sign raw digest callback.

	Name
int	**wc_PKCS7_EncodeAuthEnvelopedData**(wc_PKCS7 * pkcs7, byte * output, word32 outputSz)Encodes authenticated enveloped data.
int	**wc_PKCS7_DecodeAuthEnvelopedData**(wc_PKCS7 * pkcs7, byte * pkiMsg, word32 pkiMsgSz, byte * output, word32 outputSz)Decodes authenticated enveloped data.
int	**wc_PKCS7_EncodeEncryptedData**(wc_PKCS7 * pkcs7, byte * output, word32 outputSz)Encodes encrypted data.
int	**wc_PKCS7_SetDecodeEncryptedCb**(wc_PKCS7 * pkcs7, CallbackDecryptContent decryptionCb)Sets decode encrypted callback.
int	**wc_PKCS7_SetDecodeEncryptedCtx**(wc_PKCS7 * pkcs7, void * ctx)Sets decode encrypted context.
int	**wc_PKCS7_SetStreamMode**(wc_PKCS7 * pkcs7, byte flag, CallbackGetContent getContentCb, CallbackStreamOut streamOutCb, void * ctx)Sets stream mode for PKCS7.
int	**wc_PKCS7_GetStreamMode**(wc_PKCS7 * pkcs7)Gets stream mode setting.
int	**wc_PKCS7_SetNoCerts**(wc_PKCS7 * pkcs7, byte flag)Sets no certificates flag.
int	**wc_PKCS7_GetNoCerts**(wc_PKCS7 * pkcs7)Gets no certificates flag.
int	**wc_PKCS7_EncodeCompressedData**(wc_PKCS7 * pkcs7, byte * output, word32 outputSz)Encodes compressed data.
int	**wc_PKCS7_DecodeCompressedData**(wc_PKCS7 * pkcs7, byte * pkiMsg, word32 pkiMsgSz, byte * output, word32 outputSz)Decodes compressed data.

B.42.3 Types Documentation

```
typedef int(* CallbackAESKeyWrapUnwrap) (const byte *key, word32 keySz, const
    byte *in, word32 inSz, int wrap, byte *out, word32 outSz);
```

Callback used for a custom AES key wrap/unwrap operation.

Parameters:

- **key** Specify the key to use.
- **keySz** Size of the key to use.
- **in** Specify the input data to wrap/unwrap.
- **inSz** Size of the input data.
- **wrap** 1 if the requested operation is a key wrap, 0 for unwrap.
- **out** Specify the output buffer.
- **outSz** Size of the output buffer.

Return: The size of the wrapped/unwrapped key written to the output buffer should be returned on success. A 0 return value or error code (< 0) indicates a failure.

B.42.4 Functions Documentation

B.42.3.1 typedef CallbackAESKeyWrapUnwrap

```
int wc_PKCS7_InitWithCert(
    wc_PKCS7 * pkcs7,
    byte * der,
    word32 derSz
)
```

This function initializes a PKCS7 structure with a DER-formatted certificate. To initialize an empty PKCS7 structure, one can pass in a NULL cert and 0 for certSz.

Parameters:

- **pkcs7** pointer to the PKCS7 structure in which to store the decoded cert
- **der** pointer to a buffer containing a DER formatted ASN.1 certificate with which to initialize the PKCS7 structure
- **derSz** size of the certificate buffer

See: wc_PKCS7_Free

Return:

- 0 Returned on successfully initializing the PKCS7 structure
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_PARSE_E Returned if there is an error parsing the cert header
- ASN_OBJECT_ID_E Returned if there is an error parsing the encryption type from the cert
- ASN_EXPECT_0_E Returned if there is a formatting error in the encryption specification of the cert file
- ASN_BEFORE_DATE_E Returned if the date is before the certificate start date
- ASN_AFTER_DATE_E Returned if the date is after the certificate expiration date
- ASN_BITSTR_E Returned if there is an error parsing a bit string from the certificate
- ECC_CURVE_OID_E Returned if there is an error parsing the ECC key from the certificate
- ASN_UNKNOWN_OID_E Returned if the certificate is using an unknown key object id
- ASN_VERSION_E Returned if the ALLOW_V1_EXTENSIONS option is not defined and the certificate is a V1 or V2 certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension
- ASN_CRIT_EXT_E Returned if an unfamiliar critical extension is encountered in processing the certificate
- ASN_SIG_OID_E Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
- ASN_SIG_CONFIRM_E Returned if confirming the certification signature fails
- ASN_NAME_INVALID_E Returned if the certificate's name is not permitted by the CA name constraints
- ASN_NO_SIGNER_E Returned if there is no CA signer to verify the certificate's authenticity

Example

```
wc_PKCS7 pkcs7;
byte derBuff[] = { }; // initialize with DER-encoded certificate
if ( wc_PKCS7_InitWithCert(&pkcs7, derBuff, sizeof(derBuff)) != 0 ) {
    // error parsing certificate into pkcs7 format
}
```

```
void wc_PKCS7_Free(
    wc_PKCS7 * pkcs7
)
```

This function releases any memory allocated by a PKCS7 initializer.

Parameters:

- **pkcs7** pointer to the PKCS7 structure to free

See: wc_PKCS7_InitWithCert

Return: none No returns.

Example

```
PKCS7 pkcs7;
// initialize and use PKCS7 object

wc_PKCS7_Free(pkcs7);
```

```
int wc_PKCS7_EncodeData(
    wc_PKCS7 * pkcs7,
    byte * output,
    word32 outputSz
)
```

This function builds the PKCS7 data content type, encoding the PKCS7 structure into a buffer containing a parsable PKCS7 data packet.

Parameters:

- **pkcs7** pointer to the PKCS7 structure to encode
- **output** pointer to the buffer in which to store the encoded certificate
- **outputSz** size available in the output buffer

See: wc_PKCS7_InitWithCert

Return:

- Success On successfully encoding the PKCS7 data into the buffer, returns the index parsed up to in the PKCS7 structure. This index also corresponds to the bytes written to the output buffer.
- BUFFER_E Returned if the given buffer is not large enough to hold the encoded certificate

Example

```
PKCS7 pkcs7;
int ret;

byte derBuff[] = { }; // initialize with DER-encoded certificate
byte pkcs7Buff[FOURK_BUF];

wc_PKCS7_InitWithCert(&pkcs7, derBuff, sizeof(derBuff));
// update message and data to encode
pkcs7.privateKey = key;
pkcs7.privateKeySz = keySz;
pkcs7.content = data;
pkcs7.contentSz = dataSz;
... etc.

ret = wc_PKCS7_EncodeData(&pkcs7, pkcs7Buff, sizeof(pkcs7Buff));
if ( ret != 0 ) {
    // error encoding into output buffer
}
```

```
int wc_PKCS7_EncodeSignedData(
    wc_PKCS7 * pkcs7,
    byte * output,
    word32 outputSz
)
```

This function builds the PKCS7 signed data content type, encoding the PKCS7 structure into a buffer containing a parsable PKCS7 signed data packet. For RSA-PSS signers (WC_RSA_PSS), see PKCS#7 RSA-PSS (CMS).

Parameters:

- **pkcs7** pointer to the PKCS7 structure to encode
- **output** pointer to the buffer in which to store the encoded certificate
- **outputSz** size available in the output buffer

See:

- wc_PKCS7_InitWithCert
- wc_PKCS7_VerifySignedData

Return:

- Success On successfully encoding the PKCS7 data into the buffer, returns the index parsed up to in the PKCS7 structure. This index also corresponds to the bytes written to the output buffer.
- BAD_FUNC_ARG Returned if the PKCS7 structure is missing one or more required elements to generate a signed data packet
- MEMORY_E Returned if there is an error allocating memory
- PUBLIC_KEY_E Returned if there is an error parsing the public key

- RSA_BUFFER_E Returned if buffer error, output too small or input too large
- BUFFER_E Returned if the given buffer is not large enough to hold the encoded certificate
- MP_INIT_E may be returned if there is an error generating the signature
- MP_READ_E may be returned if there is an error generating the signature
- MP_CMP_E may be returned if there is an error generating the signature
- MP_INVMOD_E may be returned if there is an error generating the signature
- MP_EXPTMOD_E may be returned if there is an error generating the signature
- MP_MOD_E may be returned if there is an error generating the signature
- MP_MUL_E may be returned if there is an error generating the signature
- MP_ADD_E may be returned if there is an error generating the signature
- MP_MULMOD_E may be returned if there is an error generating the signature
- MP_TO_E may be returned if there is an error generating the signature
- MP_MEM may be returned if there is an error generating the signature

Example

```
PKCS7 pkcs7;
int ret;

byte data[] = {}; // initialize with data to sign
byte derBuff[] = { }; // initialize with DER-encoded certificate
byte pkcs7Buff[FOURK_BUF];

wc_PKCS7_InitWithCert(&pkcs7, derBuff, sizeof(derBuff));
// update message and data to encode
pkcs7.privateKey = key;
pkcs7.privateKeySz = keySz;
pkcs7.content = data;
pkcs7.contentSz = dataSz;
pkcs7.hashOID = SHAh;
pkcs7.rng = &rng;
... etc.

ret = wc_PKCS7_EncodeSignedData(&pkcs7, pkcs7Buff, sizeof(pkcs7Buff));
if ( ret != 0 ) {
    // error encoding into output buffer
}

wc_PKCS7_Free(&pkcs7);
```

```
int wc_PKCS7_EncodeSignedData_ex(
    wc_PKCS7 * pkcs7,
    const byte * hashBuf,
    word32 hashSz,
    byte * outputHead,
    word32 * outputHeadSz,
    byte * outputFoot,
    word32 * outputFootSz
)
```

This function builds the PKCS7 signed data content type, encoding the PKCS7 structure into a header and footer buffer containing a parsable PKCS7 signed data packet. This does not include the content.

A hash must be computed and provided for the data.

Parameters:

- **pkcs7** pointer to the PKCS7 structure to encode
- **hashBuf** pointer to computed hash for the content data
- **hashSz** size of the digest
- **outputHead** pointer to the buffer in which to store the encoded certificate header
- **outputHeadSz** pointer populated with size of output header buffer and returns actual size
- **outputFoot** pointer to the buffer in which to store the encoded certificate footer
- **outputFootSz** pointer populated with size of output footer buffer and returns actual size

See:

- wc_PKCS7_InitWithCert
- wc_PKCS7_VerifySignedData_ex

Return:

- 0=Success
- BAD_FUNC_ARG Returned if the PKCS7 structure is missing one or more required elements to generate a signed data packet
- MEMORY_E Returned if there is an error allocating memory
- PUBLIC_KEY_E Returned if there is an error parsing the public key
- RSA_BUFFER_E Returned if buffer error, output too small or input too large
- BUFFER_E Returned if the given buffer is not large enough to hold the encoded certificate
- MP_INIT_E may be returned if there is an error generating the signature
- MP_READ_E may be returned if there is an error generating the signature
- MP_CMP_E may be returned if there is an error generating the signature
- MP_INVMOD_E may be returned if there is an error generating the signature
- MP_EXPTMOD_E may be returned if there is an error generating the signature
- MP_MOD_E may be returned if there is an error generating the signature
- MP_MUL_E may be returned if there is an error generating the signature
- MP_ADD_E may be returned if there is an error generating the signature
- MP_MULMOD_E may be returned if there is an error generating the signature
- MP_TO_E may be returned if there is an error generating the signature
- MP_MEM may be returned if there is an error generating the signature

Example

```
PKCS7 pkcs7;
int ret;
byte derBuff[] = { }; // initialize with DER-encoded certificate
byte data[] = {}; // initialize with data to sign
byte pkcs7HeadBuff[FOURK_BUF/2];
byte pkcs7FootBuff[FOURK_BUF/2];
word32 pkcs7HeadSz = (word32)sizeof(pkcs7HeadBuff);
word32 pkcs7FootSz = (word32)sizeof(pkcs7HeadBuff);
enum wc_HashType hashType = WC_HASH_TYPE_SHA;
byte   hashBuf[WC_MAX_DIGEST_SIZE];
word32 hashSz = wc_HashGetDigestSize(hashType);
```

```
wc_PKCS7_InitWithCert(&pkcs7, derBuff, sizeof(derBuff));
// update message and data to encode
pkcs7.privateKey = key;
pkcs7.privateKeySz = keySz;
pkcs7.content = NULL;
pkcs7.contentSz = dataSz;
pkcs7.hashOID = SHAh;
pkcs7.rng = &rng;
... etc.

// calculate hash for content
ret = wc_HashInit(&hash, hashType);
if (ret == 0) {
    ret = wc_HashUpdate(&hash, hashType, data, sizeof(data));
    if (ret == 0) {
        ret = wc_HashFinal(&hash, hashType, hashBuf);
    }
    wc_HashFree(&hash, hashType);
}

ret = wc_PKCS7_EncodeSignedData_ex(&pkcs7, hashBuf, hashSz, pkcs7HeadBuff,
    &pkcs7HeadSz, pkcs7FootBuff, &pkcs7FootSz);
if ( ret != 0 ) {
    // error encoding into output buffer
}

wc_PKCS7_Free(&pkcs7);



int wc_PKCS7_VerifySignedData(
    wc_PKCS7 * pkcs7,
    byte * pkiMsg,
    word32 pkiMsgSz
)
```

This function takes in a transmitted PKCS7 signed data message, extracts the certificate list and certificate revocation list, and then verifies the signature. It stores the extracted content in the given PKCS7 structure.

Parameters:

- **pkcs7** pointer to the PKCS7 structure in which to store the parsed certificates
- **pkiMsg** pointer to the buffer containing the signed message to verify and decode
- **pkiMsgSz** size of the signed message

See:

- wc_PKCS7_InitWithCert
- wc_PKCS7_EncodeSignedData

Return:

- 0 Returned on successfully extracting the information from the message
- BAD_FUNC_ARG Returned if one of the input parameters is invalid
- ASN_PARSE_E Returned if there is an error parsing the given pkiMsg
- PKCS7_OID_E Returned if the given pkiMsg is not a signed data type
- ASN_VERSION_E Returned if the PKCS7 signer info is not version 1
- MEMORY_E Returned if there is an error allocating memory
- PUBLIC_KEY_E Returned if there is an error parsing the public key
- RSA_BUFFER_E Returned if buffer error, output too small or input too large
- BUFFER_E Returned if the given buffer is not large enough to hold the encoded certificate
- MP_INIT_E may be returned if there is an error generating the signature
- MP_READ_E may be returned if there is an error generating the signature
- MP_CMP_E may be returned if there is an error generating the signature
- MP_INVMOD_E may be returned if there is an error generating the signature
- MP_EXPTMOD_E may be returned if there is an error generating the signature
- MP_MOD_E may be returned if there is an error generating the signature
- MP_MUL_E may be returned if there is an error generating the signature
- MP_ADD_E may be returned if there is an error generating the signature
- MP_MULMOD_E may be returned if there is an error generating the signature
- MP_TO_E may be returned if there is an error generating the signature
- MP_MEM may be returned if there is an error generating the signature

Example

```
PKCS7 pkcs7;
int ret;
byte pkcs7Buff[] = {}; // the PKCS7 signature

wc_PKCS7_InitWithCert(&pkcs7, NULL, 0);
// update message and data to encode
pkcs7.privateKey = key;
pkcs7.privateKeySz = keySz;
pkcs7.content = data;
pkcs7.contentSz = dataSz;
... etc.

ret = wc_PKCS7_VerifySignedData(&pkcs7, pkcs7Buff, sizeof(pkcs7Buff));
if ( ret != 0 ) {
    // error encoding into output buffer
}

wc_PKCS7_Free(&pkcs7);
```

```
int wc_PKCS7_VerifySignedData_ex(
    wc_PKCS7 * pkcs7,
    const byte * hashBuf,
    word32 hashSz,
    byte * pkiMsgHead,
    word32 pkiMsgHeadSz,
    byte * pkiMsgFoot,
    word32 pkiMsgFootSz
)
```

This function takes in a transmitted PKCS7 signed data message as hash/header/footer, then extracts the certificate list and certificate revocation list, and then verifies the signature. It stores the extracted content in the given PKCS7 structure.

Parameters:

- **pkcs7** pointer to the PKCS7 structure in which to store the parsed certificates
- **hashBuf** pointer to computed hash for the content data
- **hashSz** size of the digest
- **pkiMsgHead** pointer to the buffer containing the signed message header to verify and decode
- **pkiMsgHeadSz** size of the signed message header
- **pkiMsgFoot** pointer to the buffer containing the signed message footer to verify and decode
- **pkiMsgFootSz** size of the signed message footer

See:

- wc_PKCS7_InitWithCert
- wc_PKCS7_EncodeSignedData_ex

Return:

- 0 Returned on successfully extracting the information from the message
- BAD_FUNC_ARG Returned if one of the input parameters is invalid
- ASN_PARSE_E Returned if there is an error parsing the given pkiMsg
- PKCS7_OID_E Returned if the given pkiMsg is not a signed data type
- ASN_VERSION_E Returned if the PKCS7 signer info is not version 1
- MEMORY_E Returned if there is an error allocating memory
- PUBLIC_KEY_E Returned if there is an error parsing the public key
- RSA_BUFFER_E Returned if buffer error, output too small or input too large
- BUFFER_E Returned if the given buffer is not large enough to hold the encoded certificate
- MP_INIT_E may be returned if there is an error generating the signature
- MP_READ_E may be returned if there is an error generating the signature
- MP_CMP_E may be returned if there is an error generating the signature
- MP_INVMOD_E may be returned if there is an error generating the signature
- MP_EXPTMOD_E may be returned if there is an error generating the signature
- MP_MOD_E may be returned if there is an error generating the signature
- MP_MUL_E may be returned if there is an error generating the signature
- MP_ADD_E may be returned if there is an error generating the signature
- MP_MULMOD_E may be returned if there is an error generating the signature
- MP_TO_E may be returned if there is an error generating the signature
- MP_MEM may be returned if there is an error generating the signature

Example

```
PKCS7 pkcs7;
int ret;
byte data[] = {}; // initialize with data to sign
byte pkcs7HeadBuff[] = {}; // initialize with PKCS7 header
byte pkcs7FootBuff[] = {}; // initialize with PKCS7 footer
enum wc_HashType hashType = WC_HASH_TYPE_SHA;
byte   hashBuf[WC_MAX_DIGEST_SIZE];
word32 hashSz = wc_HashGetDigestSize(hashType);
```

```
wc_PKCS7_InitWithCert(&pkcs7, NULL, 0);
// update message and data to encode
pkcs7.privateKey = key;
pkcs7.privateKeySz = keySz;
pkcs7.content = NULL;
pkcs7.contentSz = dataSz;
pkcs7.rng = &rng;
... etc.

// calculate hash for content
ret = wc_HashInit(&hash, hashType);
if (ret == 0) {
    ret = wc_HashUpdate(&hash, hashType, data, sizeof(data));
    if (ret == 0) {
        ret = wc_HashFinal(&hash, hashType, hashBuf);
    }
    wc_HashFree(&hash, hashType);
}

ret = wc_PKCS7_VerifySignedData_ex(&pkcs7, hashBuf, hashSz, pkcs7HeadBuff,
    sizeof(pkcs7HeadBuff), pkcs7FootBuff, sizeof(pkcs7FootBuff));
if ( ret != 0 ) {
    // error encoding into output buffer
}

wc_PKCS7_Free(&pkcs7);
```

```
int wc_PKCS7_SetAESKeyWrapUnwrapCb(
    wc_PKCS7 * pkcs7,
    CallbackAESKeyWrapUnwrap aesKeyWrapCb
)
```

Set the callback function to be used to perform a custom AES key wrap/unwrap operation.

Parameters:

- **pkcs7** pointer to the PKCS7 structure
- **aesKeyWrapCb** pointer to custom AES key wrap/unwrap function

Returns:

- **0** Callback function was set successfully
- **BAD_FUNC_ARG** Parameter pkcs7 is NULL

```
int wc_PKCS7_EncodeEnvelopedData(
    wc_PKCS7 * pkcs7,
    byte * output,
    word32 outputSz
)
```

This function builds the PKCS7 enveloped data content type, encoding the PKCS7 structure into a buffer containing a parsable PKCS7 enveloped data packet.

Parameters:

- **pkcs7** pointer to the PKCS7 structure to encode
- **output** pointer to the buffer in which to store the encoded certificate
- **outputSz** size available in the output buffer

See:

- wc_PKCS7_InitWithCert
- wc_PKCS7_DecodeEnvelopedData

Return:

- Success Returned on successfully encoding the message in enveloped data format, returns the size written to the output buffer
- BAD_FUNC_ARG: Returned if one of the input parameters is invalid, or if the PKCS7 structure is missing required elements
- ALGO_ID_E Returned if the PKCS7 structure is using an unsupported algorithm type. Currently, only DESb and DES3b are supported
- BUFFER_E Returned if the given output buffer is too small to store the output data
- MEMORY_E Returned if there is an error allocating memory
- RNG_FAILURE_E Returned if there is an error initializing the random number generator for encryption
- DRBG_FAILED Returned if there is an error generating numbers with the random number generator used for encryption
- NOT_COMPILED_IN may be returned if using an ECC key and wolfssl was built without HAVE_X963_KDF support

Example

```
PKCS7 pkcs7;
int ret;

byte derBuff[] = { }; // initialize with DER-encoded certificate
byte pkcs7Buff[FOURK_BUF];

wc_PKCS7_InitWithCert(&pkcs7, derBuff, sizeof(derBuff));
// update message and data to encode
pkcs7.privateKey = key;
pkcs7.privateKeySz = keySz;
pkcs7.content = data;
pkcs7.contentSz = dataSz;
... etc.

ret = wc_PKCS7_EncodeEnvelopedData(&pkcs7, pkcs7Buff, sizeof(pkcs7Buff));
if ( ret < 0 ) {
    // error encoding into output buffer
}
```

```
int wc_PKCS7_DecodeEnvelopedData(
    wc_PKCS7 * pkcs7,
    byte * pkiMsg,
    word32 pkiMsgSz,
    byte * output,
    word32 outputSz
)
```

This function unwraps and decrypts a PKCS7 enveloped data content type, decoding the message into output. It uses the private key of the PKCS7 object passed in to decrypt the message.

Parameters:

- **pkcs7** pointer to the PKCS7 structure containing the private key with which to decode the enveloped data package
- **pkiMsg** pointer to the buffer containing the enveloped data package
- **pkiMsgSz** size of the enveloped data package
- **output** pointer to the buffer in which to store the decoded message
- **outputSz** size available in the output buffer

See:

- wc_PKCS7_InitWithCert
- wc_PKCS7_EncodeEnvelopedData

Return:

- On successfully extracting the information from the message, returns the bytes written to output
- BAD_FUNC_ARG Returned if one of the input parameters is invalid
- ASN_PARSE_E Returned if there is an error parsing the given pkiMsg
- PKCS7_OID_E Returned if the given pkiMsg is not an enveloped data type
- ASN_VERSION_E Returned if the PKCS7 signer info is not version 0
- MEMORY_E Returned if there is an error allocating memory
- ALGO_ID_E Returned if the PKCS7 structure is using an unsupported algorithm type. Currently, only DESb and DES3b are supported for encryption, with RSAk for signature generation
- PKCS7_RECIP_E Returned if there is no recipient found in the enveloped data that matches the recipient provided
- RSA_BUFFER_E Returned if there is an error during RSA signature verification due to buffer error, output too small or input too large.
- MP_INIT_E may be returned if there is an error during signature verification
- MP_READ_E may be returned if there is an error during signature verification
- MP_CMP_E may be returned if there is an error during signature verification
- MP_INVMOD_E may be returned if there is an error during signature verification
- MP_EXPTMOD_E may be returned if there is an error during signature verification
- MP_MOD_E may be returned if there is an error during signature verification
- MP_MUL_E may be returned if there is an error during signature verification
- MP_ADD_E may be returned if there is an error during signature verification
- MP_MULMOD_E may be returned if there is an error during signature verification
- MP_TO_E may be returned if there is an error during signature verification
- MP_MEM may be returned if there is an error during signature verification
- NOT_COMPILED_IN may be returned if the EnvelopedData is encrypted using an ECC key and wolfssl was built without HAVE_X963_KDF support

Note that if the EnvelopedData is encrypted using an ECC key and the KeyAgreementRecipientInfo structure, then either the HAVE_AES_KEYWRAP build option should be enabled to enable the wolfcrypt built-in AES key wrap/unwrap functionality, or a custom AES key wrap/unwrap callback should be set with wc_PKCS7_SetAESKeyWrapUnwrapCb(). If neither of these is true, decryption will fail.

Example

```
PKCS7 pkcs7;
byte received[] = { }; // initialize with received enveloped message
byte decoded[FOURK_BUF];
int decodedSz;

// initialize pkcs7 with certificate
// update key
pkcs7.privateKey = key;
pkcs7.privateKeySz = keySz;

decodedSz = wc_PKCS7_DecodeEnvelopedData(&pkcs7, received,
    sizeof(received),decoded, sizeof(decoded));
if ( decodedSz < 0 ) {
    // error decoding message
}
```

```
int wc_PKCS7_GetEnvelopedDataKariRid(
    const byte * in,
    word32 inSz,
    byte * out,
    word32 * outSz
)
```

This function extracts the KeyAgreeRecipientIdentifier object from an EnvelopedData package containing a KeyAgreeRecipientInfo RecipientInfo object. Only the first KeyAgreeRecipientIdentifer found in the first RecipientInfo is copied. This function does not support multiple RecipientInfo objects or multiple RecipientEncryptedKey objects within an KeyAgreeRecipientInfo.

Parameters:

- **in** Input buffer containing the EnvelopedData ContentInfo message.
- **inSz** Size of the input buffer.
- **out** Output buffer.
- **outSz** Output buffer size on input, Size written on output.

Return:

- Returns 0 on success.
- BAD_FUNC_ARG Returned if one of the input parameters is invalid.
- ASN_PARSE_E Returned if there is an error parsing the input message.
- PKCS7_OID_E Returned if the input message is not an enveloped data type.
- BUFFER_E Returned if there is not enough room in the output buffer.

```
int wc_PKCS7_DecodeEncryptedData(
    wc_PKCS7 * pkcs7,
    byte * pkiMsg,
    word32 pkiMsgSz,
    byte * output,
    word32 outputSz
)
```

This function unwraps and decrypts a PKCS7 encrypted data content type, decoding the message into output. It uses the encryption key of the PKCS7 object passed in via pkcs7->encryptionKey and pkcs7->encryptionKeySz to decrypt the message.

Parameters:

- **pkcs7** pointer to the PKCS7 structure containing the encryption key with which to decode the encrypted data package
- **pkiMsg** pointer to the buffer containing the encrypted data package
- **pkiMsgSz** size of the encrypted data package
- **output** pointer to the buffer in which to store the decoded message
- **outputSz** size available in the output buffer

See: wc_PKCS7_InitWithCert

Return:

- On successfully extracting the information from the message, returns the bytes written to output
- BAD_FUNC_ARG Returned if one of the input parameters is invalid
- ASN_PARSE_E Returned if there is an error parsing the given pkiMsg
- PKCS7_OID_E Returned if the given pkiMsg is not an encrypted data type
- ASN_VERSION_E Returned if the PKCS7 signer info is not version 0
- MEMORY_E Returned if there is an error allocating memory
- BUFFER_E Returned if the encrypted content size is invalid

Example

```
PKCS7 pkcs7;
byte received[] = { }; // initialize with received encrypted data message
byte decoded[FOURK_BUF];
int decodedSz;

// initialize pkcs7 with certificate
// update key
pkcs7.encryptionKey = key;
pkcs7.encryptionKeySz = keySz;

decodedSz = wc_PKCS7_DecodeEncryptedData(&pkcs7, received,
    sizeof(received), decoded, sizeof(decoded));
if ( decodedSz < 0 ) {
    // error decoding message
}
```

```
int wc_PKCS7_DecodeEncryptedKeyPackage(
    wc_PKCS7 * pkcs7,
    byte * pkiMsg,
    word32 pkiMsgSz,
    byte * output,
    word32 outputSz
)
```

This function unwraps and decrypts a PKCS7 encrypted key package content type, decoding the message into output. If the wrapped content type is EncryptedData, the encryption key must be set in the pkcs7 input structure (via pkcs7->encryptionKey and pkcs7->encryptionKeySz). If the wrapped content type is EnvelopedData, the private key must be set in the pkcs7 input structure (via pkcs7->privateKey and pkcs7->privateKeySz). A wrapped content type of AuthEnvelopedData is not currently supported.

Parameters:

- **pkcs7** pointer to the PKCS7 structure containing the private key or encryption key with which to decode the encrypted key package
- **pkiMsg** pointer to the buffer containing the encrypted key package message
- **pkiMsgSz** size of the encrypted key package message
- **output** pointer to the buffer in which to store the decoded output
- **outputSz** size available in the output buffer

See: wc_PKCS7_InitWithCert

Return:

- On successfully extracting the information from the message, returns the bytes written to output
- BAD_FUNC_ARG Returned if one of the input parameters is invalid
- ASN_PARSE_E Returned if there is an error parsing the given pkiMsg or if the wrapped content type is EncryptedData and support for EncryptedData is not compiled in (e.g. NO_PKCS7_ENCRYPTED_DATA is set)
- PKCS7_OID_E Returned if the given pkiMsg is not an encrypted key package data type

This function will automatically call either wc_PKCS7_DecodeEnvelopedData() or wc_PKCS7_DecodeEncryptedData() depending on the wrapped content type. This function could also return any error code from either of those functions in addition to the error codes listed here.

Example

```
PKCS7 pkcs7;
byte received[] = { }; // initialize with received encrypted data message
byte decoded[FOURK_BUF];
int decodedSz;

// initialize pkcs7 with certificate
// update key for expected EnvelopedData (example)
pkcs7.privateKey = key;
pkcs7.privateKeySz = keySz;

decodedSz = wc_PKCS7_DecodeEncryptedKeyPackage(&pkcs7, received,
    sizeof(received), decoded, sizeof(decoded));
if ( decodedSz < 0 ) {
```

```
    // error decoding message
}



int wc_PKCS7_DecodeSymmetricKeyPackageAttribute(
    const byte * skp,
    word32 skpSz,
    size_t index,
    const byte ** attr,
    word32 * attrSz
)
```

This function provides access to a SymmetricKeyPackage attribute.

Parameters:

- **skp** Input buffer containing the SymmetricKeyPackage object.
- **skpSz** Size of the SymmetricKeyPackage object.
- **index** Index of the attribute to access.
- **attr** Buffer in which to store the pointer to the requested attribute object.
- **attrSz** Buffer in which to store the size of the requested attribute object.

Return:

- 0 The requested attribute has been successfully located. attr and attrSz output variables are populated with the address and size of the attribute. The attribute will be in the same buffer passed in via the skp input pointer.
- BAD_FUNC_ARG One of the input parameters is invalid.
- ASN_PARSE_E An error was encountered parsing the input object.
- BAD_INDEX_E The requested attribute index was invalid.

```
int wc_PKCS7_DecodeSymmetricKeyPackageKey(
    const byte * skp,
    word32 skpSz,
    size_t index,
    const byte ** key,
    word32 * keySz
)
```

This function provides access to a SymmetricKeyPackage key.

Parameters:

- **skp** Input buffer containing the SymmetricKeyPackage object.
- **skpSz** Size of the SymmetricKeyPackage object.
- **index** Index of the key to access.
- **key** Buffer in which to store the pointer to the requested key object.
- **keySz** Buffer in which to store the size of the requested key object.

Return:

- 0 The requested key has been successfully located. key and keySz output variables are populated with the address and size of the key. The key will be in the same buffer passed in via the skp input pointer.
- BAD_FUNC_ARG One of the input parameters is invalid.
- ASN_PARSE_E An error was encountered parsing the input object.
- BAD_INDEX_E The requested key index was invalid.

```
int wc_PKCS7_DecodeOneSymmetricKeyAttribute(
    const byte * osk,
    word32 oskSz,
    size_t index,
    const byte ** attr,
    word32 * attrSz
)
```

This function provides access to a OneSymmetricKey attribute.

Parameters:

- **osk** Input buffer containing the OneSymmetricKey object.
- **oskSz** Size of the OneSymmetricKey object.
- **index** Index of the attribute to access.
- **attr** Buffer in which to store the pointer to the requested attribute object.
- **attrSz** Buffer in which to store the size of the requested attribute object.

Return:

- 0 The requested attribute has been successfully located. attr and attrSz output variables are populated with the address and size of the attribute. The attribute will be in the same buffer passed in via the osk input pointer.
- BAD_FUNC_ARG One of the input parameters is invalid.
- ASN_PARSE_E An error was encountered parsing the input object.
- BAD_INDEX_E The requested attribute index was invalid.

```
int wc_PKCS7_DecodeOneSymmetricKeyKey(
    const byte * osk,
    word32 oskSz,
    const byte ** key,
    word32 * keySz
)
```

This function provides access to a OneSymmetricKey key.

Parameters:

- **osk** Input buffer containing the OneSymmetricKey object.

- **oskSz** Size of the OneSymmetricKey object.
- **key** Buffer in which to store the pointer to the requested key object.
- **keySz** Buffer in which to store the size of the requested key object.

Return:

- 0 The requested key has been successfully located. key and keySz output variables are populated with the address and size of the key. The key will be in the same buffer passed in via the osk input pointer.
- BAD_FUNC_ARG One of the input parameters is invalid.
- ASN_PARSE_E An error was encountered parsing the input object.

```
PKCS7 * wolfSSL_PKCS7_new(
    void
)
```

Creates new PKCS7 structure.

Parameters:

- **none** No parameters

See: wolfSSL_PKCS7_free

Return:

- Pointer to new PKCS7 structure on success
- NULL on error

Example

```
PKCS7* pkcs7 = wolfSSL_PKCS7_new();
if (pkcs7 != NULL) {
    // use pkcs7
    wolfSSL_PKCS7_free(pkcs7);
}
```

```
PKCS7_SIGNED * wolfSSL_PKCS7_SIGNED_new(
    void
)
```

Creates new PKCS7_SIGNED structure.

Parameters:

- **none** No parameters

See: wolfSSL_PKCS7_SIGNED_free

Return:

- Pointer to new PKCS7_SIGNED structure on success
- NULL on error

Example

```
PKCS7_SIGNED* p7 = wolfSSL_PKCS7_SIGNED_new();
if (p7 != NULL) {
    // use p7
    wolfSSL_PKCS7_SIGNED_free(p7);
}
```

```
void wolfSSL_PKCS7_free(
    PKCS7 * p7
)
```

Frees PKCS7 structure.

Parameters:

- **p7** PKCS7 structure to free

See: wolfSSL_PKCS7_new

Return: none No returns

Example

```
PKCS7* pkcs7 = wolfSSL_PKCS7_new();
wolfSSL_PKCS7_free(pkcs7);
```

```
void wolfSSL_PKCS7_SIGNED_free(
    PKCS7_SIGNED * p7
)
```

Frees PKCS7_SIGNED structure.

Parameters:

- **p7** PKCS7_SIGNED structure to free

See: wolfSSL_PKCS7_SIGNED_new

Return: none No returns

Example

```
PKCS7_SIGNED* p7 = wolfSSL_PKCS7_SIGNED_new();
wolfSSL_PKCS7_SIGNED_free(p7);
```

```
PKCS7 * wolfSSL_d2i_PKCS7(
    PKCS7 ** p7,
    const unsigned char ** in,
    int len
)
```

Decodes DER-encoded PKCS7 structure.

Parameters:

- **p7** Pointer to PKCS7 pointer (can be NULL)
- **in** Pointer to DER-encoded data
- **len** Length of DER data

See: wolfSSL_i2d_PKCS7

Return:

- Pointer to decoded PKCS7 structure on success
- NULL on error

Example

```
PKCS7* p7 = NULL;
const unsigned char* der = ...; // DER data
p7 = wolfSSL_d2i_PKCS7(&p7, &der, derLen);
```

```
PKCS7 * wolfSSL_d2i_PKCS7_bio(
    WOLFSSL_BIO * bio,
    PKCS7 ** p7
)
```

Decodes PKCS7 from BIO.

Parameters:

- **bio** BIO to read from
- **p7** Pointer to PKCS7 pointer (can be NULL)

See: wolfSSL_i2d_PKCS7_bio

Return:

- Pointer to decoded PKCS7 structure on success
- NULL on error

Example

```
WOLFSSL_BIO* bio = wolfSSL_BIO_new_file("pkcs7.der", "rb");
PKCS7* p7 = wolfSSL_d2i_PKCS7_bio(bio, NULL);
```

```
int wolfSSL_i2d_PKCS7_bio(
    WOLFSSL_BIO * bio,
    PKCS7 * p7
)
```

Encodes PKCS7 to BIO.

Parameters:

- **bio** BIO to write to
- **p7** PKCS7 structure to encode

See: wolfSSL_d2i_PKCS7_bio

Return:

- Length written on success
- negative on error

Example

```
WOLFSSL_BIO* bio = wolfSSL_BIO_new(wolfSSL_BIO_s_mem());
int ret = wolfSSL_i2d_PKCS7_bio(bio, p7);
```

```
int wolfSSL_i2d_PKCS7(
    PKCS7 * p7,
    unsigned char ** out
)
```

Encodes PKCS7 to DER.

Parameters:

- **p7** PKCS7 structure to encode
- **out** Pointer to output buffer pointer

See: wolfSSL_d2i_PKCS7

Return:

- Length written on success
- negative on error

Example

```
unsigned char* der = NULL;
int len = wolfSSL_i2d_PKCS7(p7, &der);
```

```
PKCS7 * wolfSSL_PKCS7_sign(
    WOLFSSL_X509 * signer,
    WOLFSSL_EVP_PKEY * pkey,
    WOLFSSL_STACK * certs,
    WOLFSSL_BIO * in,
    int flags
)
```

Creates signed PKCS7 message.

Parameters:

- **signer** Signer certificate
- **pkey** Private key
- **certs** Additional certificates
- **in** Input data BIO
- **flags** Operation flags

See: wolfSSL_PKCS7_verify

Return:

- Pointer to signed PKCS7 structure on success
- NULL on error

Example

```
PKCS7* p7 = wolfSSL_PKCS7_sign(cert, pkey, NULL, bio, 0);
```

```
int wolfSSL_PKCS7_verify(
    PKCS7 * p7,
    WOLFSSL_STACK * certs,
    WOLFSSL_X509_STORE * store,
    WOLFSSL_BIO * in,
    WOLFSSL_BIO * out,
    int flags
)
```

Verifies signed PKCS7 message.

Parameters:

- **p7** PKCS7 structure to verify

- **certs** Certificate stack
- **store** Certificate store
- **in** Input data BIO
- **out** Output BIO
- **flags** Operation flags

See: wolfSSL_PKCS7_sign

Return:

- 1 on success
- 0 or negative on error

Example

```
int ret = wolfSSL_PKCS7_verify(p7, NULL, store, NULL, out, 0);
```

```
int wolfSSL_PKCS7_final(
    PKCS7 * pkcs7,
    WOLFSSL_BIO * in,
    int flags
)
```

Finalizes PKCS7 structure with data.

Parameters:

- **pkcs7** PKCS7 structure
- **in** Input data BIO
- **flags** Operation flags

See: wolfSSL_PKCS7_sign

Return:

- 1 on success
- 0 or negative on error

Example

```
int ret = wolfSSL_PKCS7_final(pkcs7, bio, 0);
```

```
int wolfSSL_PKCS7_encode_certs(
    PKCS7 * p7,
    WOLFSSL_STACK * certs,
    WOLFSSL_BIO * out
)
```

Encodes certificates into PKCS7.

Parameters:

- **p7** PKCS7 structure
- **certs** Certificate stack
- **out** Output BIO

See: wolfSSL_PKCS7_to_stack

Return:

- 1 on success
- 0 or negative on error

Example

```
int ret = wolfSSL_PKCS7_encode_certs(p7, certs, bio);
```

```
WOLFSSL_STACK * wolfSSL_PKCS7_to_stack(
    PKCS7 * pkcs7
)
```

Converts PKCS7 certificates to stack.

Parameters:

- **pkcs7** PKCS7 structure

See: wolfSSL_PKCS7_encode_certs

Return:

- Pointer to certificate stack on success
- NULL on error

Example

```
WOLFSSL_STACK* certs = wolfSSL_PKCS7_to_stack(pkcs7);
```

```
WOLFSSL_STACK * wolfSSL_PKCS7_get0_signers(
    PKCS7 * p7,
    WOLFSSL_STACK * certs,
    int flags
)
```

Gets signer certificates from PKCS7.

Parameters:

- **p7** PKCS7 structure
- **certs** Certificate stack
- **flags** Operation flags

See: wolfSSL_PKCS7_verify

Return:

- Pointer to signer certificate stack on success
- NULL on error

Example

```
WOLFSSL_STACK* signers = wolfSSL_PKCS7_get0_signers(p7, NULL, 0);
```

```
int wolfSSL_PEM_write_bio_PKCS7(
    WOLFSSL_BIO * bio,
    PKCS7 * p7
)
```

Writes PKCS7 to BIO in PEM format.

Parameters:

- **bio** Output BIO
- **p7** PKCS7 structure

See: wolfSSL_SMIME_write_PKCS7

Return:

- 1 on success
- 0 or negative on error

Example

```
int ret = wolfSSL_PEM_write_bio_PKCS7(bio, p7);
```

```
PKCS7 * wolfSSL_SMIME_read_PKCS7(
    WOLFSSL_BIO * in,
    WOLFSSL_BIO ** bcont
)
```

Reads S/MIME PKCS7 from BIO.

Parameters:

- **in** Input BIO
- **bcont** Pointer to content BIO pointer

See: wolfSSL_SMIME_write_PKCS7

Return:

- Pointer to PKCS7 structure on success
- NULL on error

Example

```
WOLFSSL_BIO* cont = NULL;
PKCS7* p7 = wolfSSL_SMIME_read_PKCS7(bio, &cont);
```

```
int wolfSSL_SMIME_write_PKCS7(
    WOLFSSL_BIO * out,
    PKCS7 * pkcs7,
    WOLFSSL_BIO * in,
    int flags
)
```

Writes PKCS7 to BIO in S/MIME format.

Parameters:

- **out** Output BIO
- **pkcs7** PKCS7 structure
- **in** Input data BIO
- **flags** Operation flags

See: wolfSSL_SMIME_read_PKCS7

Return:

- 1 on success
- 0 or negative on error

Example

```
int ret = wolfSSL_SMIME_write_PKCS7(out, pkcs7, in, 0);
```

```
wc_PKCS7 * wc_PKCS7_New(
    void * heap,
    int devId
)
```

Creates new wc_PKCS7 structure.

Parameters:

- **heap** Heap hint
- **devId** Device ID

See: wc_PKCS7_Init

Return:

- Pointer to new wc_PKCS7 structure on success
- NULL on error

Example

```
wc_PKCS7* pkcs7 = wc_PKCS7_New(NULL, INVALID_DEVID);
```

```
void wc_PKCS7_SetUnknownExtCallback(
    wc_PKCS7 * pkcs7,
    wc_UnknownExtCallback cb
)
```

Sets unknown extension callback.

Parameters:

- **pkcs7** PKCS7 structure
- **cb** Callback function

See: wc_PKCS7_Init

Return: none No returns

Example

```
wc_PKCS7_SetUnknownExtCallback(pkcs7, myCallback);
```

```
int wc_PKCS7_Init(
    wc_PKCS7 * pkcs7,
    void * heap,
    int devId
)
```

Initializes wc_PKCS7 structure.

Parameters:

- **pkcs7** PKCS7 structure
- **heap** Heap hint
- **devId** Device ID

See: wc_PKCS7_New

Return:

- 0 on success
- negative on error

Example

```
wc_PKCS7 pkcs7;
int ret = wc_PKCS7_Init(&pkcs7, NULL, INVALID_DEVID);
```

```
int wc_PKCS7_AddCertificate(
    wc_PKCS7 * pkcs7,
    byte * der,
    word32 derSz
)
```

Adds certificate to PKCS7.

Parameters:

- **pkcs7** PKCS7 structure
- **der** DER-encoded certificate
- **derSz** Certificate size

See: wc_PKCS7_Init

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_AddCertificate(&pkcs7, cert, certSz);
```

```
int wc_PKCS7_GetAttributeValue(
    wc_PKCS7 * pkcs7,
    const byte * oid,
    word32 oidSz,
    byte * out,
    word32 * outSz
)
```

Gets attribute value from PKCS7.

Parameters:

- **pkcs7** PKCS7 structure
- **oid** Attribute OID
- **oidSz** OID size
- **out** Output buffer
- **outSz** Output buffer size pointer

See: wc_PKCS7_Init

Return:

- 0 on success
- negative on error

Example

```
byte value[256];
word32 valueSz = sizeof(value);
int ret = wc_PKCS7_GetAttributeValue(&pkcs7, oid, oidSz, value,
                                     &valueSz);
```

```
int wc_PKCS7_SetSignerIdentifierType(
    wc_PKCS7 * pkcs7,
    int type
)
```

Sets signer identifier type.

Parameters:

- **pkcs7** PKCS7 structure
- **type** Identifier type

See: wc_PKCS7_Init

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_SetSignerIdentifierType(&pkcs7, CMS_SKID);
```

```
int wc_PKCS7_SetContentType(
    wc_PKCS7 * pkcs7,
    byte * contentType,
    word32 sz
)
```

Sets content type.

Parameters:

- **pkcs7** PKCS7 structure
- **contentType** Content type OID
- **sz** OID size

See: wc_PKCS7_Init

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_SetContentType(&pkcs7, DATA, sizeof(DATA));
```

```
int wc_PKCS7_GetPadSize(
    word32 inputSz,
    word32 blockSz
)
```

Gets padding size for block cipher.

Parameters:

- **inputSz** Input size
- **blockSz** Block size

See: wc_PKCS7_PadData

Return: Padding size

Example

```
int padSz = wc_PKCS7_GetPadSize(dataSz, AES_BLOCK_SIZE);
```

```
int wc_PKCS7_PadData(
    byte * in,
    word32 inSz,
    byte * out,
    word32 outSz,
    word32 blockSz
)
```

Pads data for block cipher.

Parameters:

- **in** Input data
- **inSz** Input size
- **out** Output buffer
- **outSz** Output buffer size
- **blockSz** Block size

See: wc_PKCS7_GetPadSize

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_PadData(data, dataSz, padded, paddedSz,
                           AES_BLOCK_SIZE);
```

```
int wc_PKCS7_SetCustomSKID(
    wc_PKCS7 * pkcs7,
    const byte * in,
    word16 inSz
)
```

Sets custom subject key identifier.

Parameters:

- **pkcs7** PKCS7 structure
- **in** SKID data
- **inSz** SKID size

See: wc_PKCS7_Init

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_SetCustomSKID(&pkcs7, skid, skidSz);
```

```
int wc_PKCS7_SetDetached(
    wc_PKCS7 * pkcs7,
    word16 flag
)
```

Sets detached signature flag.

Parameters:

- **pkcs7** PKCS7 structure
- **flag** Detached flag (1=detached, 0=attached)

See: wc_PKCS7_Init

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_SetDetached(&pkcs7, 1);
```

```
int wc_PKCS7_NoDefaultSignedAttribs(
    wc_PKCS7 * pkcs7
)
```

Disables default signed attributes.

Parameters:

- **pkcs7** PKCS7 structure

See: wc_PKCS7_SetDefaultSignedAttribs

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_NoDefaultSignedAttribs(&pkcs7);
```

```
int wc_PKCS7_SetDefaultSignedAttribs(
    wc_PKCS7 * pkcs7,
    word16 flag
)
```

Sets default signed attributes flag.

Parameters:

- **pkcs7** PKCS7 structure
- **flag** Default attributes flag

See: wc_PKCS7_NoDefaultSignedAttribs

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_SetDefaultSignedAttribs(&pkcs7, 1);
```

```
void wc_PKCS7_AllowDegenerate(
    wc_PKCS7 * pkcs7,
    word16 flag
)
```

Allows degenerate PKCS7 (no signers).

Parameters:

- **pkcs7** PKCS7 structure
- **flag** Allow degenerate flag

See: wc_PKCS7_Init

Return: none No returns

Example

```
wc_PKCS7_AllowDegenerate(&pkcs7, 1);
```

```
int wc_PKCS7_GetSignerSID(
    wc_PKCS7 * pkcs7,
    byte * out,
    word32 * outSz
)
```

Gets signer subject identifier.

Parameters:

- **pkcs7** PKCS7 structure
- **out** Output buffer
- **outSz** Output buffer size pointer

See: wc_PKCS7_Init

Return:

- 0 on success
- negative on error

Example

```
byte sid[256];
word32 sidSz = sizeof(sid);
int ret = wc_PKCS7_GetSignerSID(&pkcs7, sid, &sidSz);
```

```
int wc_PKCS7_EncodeSignedFPD(
    wc_PKCS7 * pkcs7,
    byte * privateKey,
    word32 privateKeySz,
    int signOID,
    int hashOID,
    byte * content,
    word32 contentSz,
    PKCS7Attrib * signedAttribs,
    word32 signedAttribsSz,
    byte * output,
    word32 outputSz
)
```

Encodes signed FirmwarePackageData.

Parameters:

- **pkcs7** PKCS7 structure
- **privateKey** Private key
- **privateKeySz** Private key size
- **signOID** Signature algorithm OID
- **hashOID** Hash algorithm OID
- **content** Content data
- **contentSz** Content size
- **signedAttribs** Signed attributes
- **signedAttribsSz** Signed attributes count
- **output** Output buffer
- **outputSz** Output buffer size

See: wc_PKCS7_EncodeSignedData

Return:

- Size of encoded data on success
- negative on error

Example

```
int ret = wc_PKCS7_EncodeSignedFPD(&pkcs7, key, keySz, RSAk, SHAh,
                                   data, dataSz, NULL, 0, out, outSz);
```

```
int wc_PKCS7_EncodeSignedEncryptedFPD(
    wc_PKCS7 * pkcs7,
    byte * encryptKey,
    word32 encryptKeySz,
    byte * privateKey,
    word32 privateKeySz,
    int encryptOID,
    int signOID,
    int hashOID,
    byte * content,
    word32 contentSz,
    PKCS7Attrib * unprotectedAttribs,
    word32 unprotectedAttribsSz,
    PKCS7Attrib * signedAttribs,
    word32 signedAttribsSz,
    byte * output,
    word32 outputSz
)
```

Encodes signed encrypted FirmwarePackageData.

Parameters:

- **pkcs7** PKCS7 structure
- **encryptKey** Encryption key
- **encryptKeySz** Encryption key size
- **privateKey** Private key
- **privateKeySz** Private key size
- **encryptOID** Encryption algorithm OID
- **signOID** Signature algorithm OID
- **hashOID** Hash algorithm OID
- **content** Content data
- **contentSz** Content size
- **unprotectedAttribs** Unprotected attributes
- **unprotectedAttribsSz** Unprotected attributes count
- **signedAttribs** Signed attributes
- **signedAttribsSz** Signed attributes count
- **output** Output buffer
- **outputSz** Output buffer size

See: wc_PKCS7_EncodeSignedFPD

Return:

- Size of encoded data on success
- negative on error

Example

```
int ret = wc_PKCS7_EncodeSignedEncryptedFPD(&pkcs7, encKey, encKeySz,
                                            key, keySz, AES256CBCb,
                                            RSAk, SHAh, data, dataSz,
                                            NULL, 0, NULL, 0, out,
                                            outSz);
```

```
int wc_PKCS7_EncodeSignedCompressedFPD(
    wc_PKCS7 * pkcs7,
    byte * privateKey,
    word32 privateKeySz,
    int signOID,
    int hashOID,
    byte * content,
    word32 contentSz,
    PKCS7Attrib * signedAttribs,
    word32 signedAttribsSz,
    byte * output,
    word32 outputSz
)
```

Encodes signed compressed FirmwarePackageData.

Parameters:

- **pkcs7** PKCS7 structure
- **privateKey** Private key
- **privateKeySz** Private key size
- **signOID** Signature algorithm OID
- **hashOID** Hash algorithm OID
- **content** Content data
- **contentSz** Content size
- **signedAttribs** Signed attributes
- **signedAttribsSz** Signed attributes count
- **output** Output buffer
- **outputSz** Output buffer size

See: wc_PKCS7_EncodeSignedFPD

Return:

- Size of encoded data on success
- negative on error

Example

```
int ret = wc_PKCS7_EncodeSignedCompressedFPD(&pkcs7, key, keySz, RSAk,
                                             SHAh, data, dataSz, NULL,
                                             0, out, outSz);
```

```
int wc_PKCS7_EncodeSignedEncryptedCompressedFPD(
    wc_PKCS7 * pkcs7,
    byte * encryptKey,
    word32 encryptKeySz,
    byte * privateKey,
    word32 privateKeySz,
    int encryptOID,
    int signOID,
    int hashOID,
    byte * content,
    word32 contentSz,
    PKCS7Attrib * unprotectedAttribs,
    word32 unprotectedAttribsSz,
    PKCS7Attrib * signedAttribs,
    word32 signedAttribsSz,
    byte * output,
    word32 outputSz
)
```

Encodes signed encrypted compressed FirmwarePackageData.

Parameters:

- **pkcs7** PKCS7 structure
- **encryptKey** Encryption key
- **encryptKeySz** Encryption key size
- **privateKey** Private key
- **privateKeySz** Private key size
- **encryptOID** Encryption algorithm OID
- **signOID** Signature algorithm OID
- **hashOID** Hash algorithm OID
- **content** Content data
- **contentSz** Content size
- **unprotectedAttribs** Unprotected attributes
- **unprotectedAttribsSz** Unprotected attributes count
- **signedAttribs** Signed attributes
- **signedAttribsSz** Signed attributes count
- **output** Output buffer
- **outputSz** Output buffer size

See: wc_PKCS7_EncodeSignedCompressedFPD

Return:

- Size of encoded data on success

- negative on error

Example

```
int ret = wc_PKCS7_EncodeSignedEncryptedCompressedFPD(&pkcs7, encKey,
                                                      encKeySz, key,
                                                      keySz, AES256CBCb,
                                                      RSAk, SHAh, data,
                                                      dataSz, NULL, 0,
                                                      NULL, 0, out,
                                                      outSz);
```

```
int wc_PKCS7_AddRecipient_KTRI(
    wc_PKCS7 * pkcs7,
    const byte * cert,
    word32 certSz,
    int options
)
```

Adds KTRI recipient.

Parameters:

- **pkcs7** PKCS7 structure
- **cert** Recipient certificate
- **certSz** Certificate size
- **options** Options flags

See: wc_PKCS7_AddRecipient_KARI

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_AddRecipient_KTRI(&pkcs7, cert, certSz, 0);
```

```
int wc_PKCS7_AddRecipient_KARI(
    wc_PKCS7 * pkcs7,
    const byte * cert,
    word32 certSz,
    int keyWrapOID,
    int keyAgreeOID,
    byte * ukm,
    word32 ukmSz,
    int options
)
```

Adds KARI recipient.

Parameters:

- **pkcs7** PKCS7 structure
- **cert** Recipient certificate
- **certSz** Certificate size
- **keyWrapOID** Key wrap algorithm OID
- **keyAgreeOID** Key agreement algorithm OID
- **ukm** User keying material
- **ukmSz** UKM size
- **options** Options flags

See: wc_PKCS7_AddRecipient_KTRI

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_AddRecipient_KARI(&pkcs7, cert, certSz, AES256_WRAP,
                                     dhSinglePass_stdDH_sha256kdf_scheme,
                                     NULL, 0, 0);
```

```
int wc_PKCS7_SetKey(
    wc_PKCS7 * pkcs7,
    byte * key,
    word32 keySz
)
```

Sets encryption key.

Parameters:

- **pkcs7** PKCS7 structure
- **key** Encryption key
- **keySz** Key size

See: wc_PKCS7_Init

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_SetKey(&pkcs7, key, keySz);
```

```
int wc_PKCS7_AddRecipient_KEKRI(
    wc_PKCS7 * pkcs7,
    int keyWrapOID,
    byte * kek,
    word32 kekSz,
    byte * keyID,
    word32 keyIdSz,
    void * timePtr,
    byte * otherOID,
    word32 otherOIDSz,
    byte * other,
    word32 otherSz,
    int options
)
```

Adds KEKRI recipient.

Parameters:

- **pkcs7** PKCS7 structure
- **keyWrapOID** Key wrap algorithm OID
- **kek** Key encryption key
- **kekSz** KEK size
- **keyID** Key identifier
- **keyIdSz** Key ID size
- **timePtr** Time pointer
- **otherOID** Other OID
- **otherOIDSz** Other OID size
- **other** Other data
- **otherSz** Other data size
- **options** Options flags

See: wc_PKCS7_AddRecipient_KTRI

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_AddRecipient_KEKRI(&pkcs7, AES256_WRAP, kek, kekSz,
                                      keyId, keyIdSz, NULL, NULL, 0,
                                      NULL, 0, 0);
```

```
int wc_PKCS7_SetPassword(
    wc_PKCS7 * pkcs7,
    byte * passwd,
    word32 pLen
```

)

Sets password for PWRI.

Parameters:

- **pkcs7** PKCS7 structure
- **passwd** Password
- **pLen** Password length

See: wc_PKCS7_AddRecipient_PWRI

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_SetPassword(&pkcs7, password, passwordLen);
```

```
int wc_PKCS7_AddRecipient_PWRI(
    wc_PKCS7 * pkcs7,
    byte * passwd,
    word32 pLen,
    byte * salt,
    word32 saltSz,
    int kdfOID,
    int prfOID,
    int iterations,
    int kekEncryptOID,
    int options
)
```

Adds PWRI recipient.

Parameters:

- **pkcs7** PKCS7 structure
- **passwd** Password
- **pLen** Password length
- **salt** Salt
- **saltSz** Salt size
- **kdfOID** KDF algorithm OID
- **prfOID** PRF algorithm OID
- **iterations** Iteration count
- **kekEncryptOID** KEK encryption algorithm OID
- **options** Options flags

See: wc_PKCS7_SetPassword

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_AddRecipient_PWRI(&pkcs7, password, passwordLen,
                                     salt, saltSz, PBKDF2_OID, HMACh,
                                     10000, AES256CBCb, 0);
```

```
int wc_PKCS7_SetOriEncryptCtx(
    wc_PKCS7 * pkcs7,
    void * ctx
)
```

Sets originator encryption context.

Parameters:

- **pkcs7** PKCS7 structure
- **ctx** Context pointer

See: wc_PKCS7_SetOriDecryptCtx

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_SetOriEncryptCtx(&pkcs7, myContext);
```

```
int wc_PKCS7_SetOriDecryptCtx(
    wc_PKCS7 * pkcs7,
    void * ctx
)
```

Sets originator decryption context.

Parameters:

- **pkcs7** PKCS7 structure
- **ctx** Context pointer

See: wc_PKCS7_SetOriEncryptCtx

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_SetOriDecryptCtx(&pkcs7, myContext);
```

```
int wc_PKCS7_SetOriDecryptCb(
    wc_PKCS7 * pkcs7,
    CallbackOriDecrypt cb
)
```

Sets originator decryption callback.

Parameters:

- **pkcs7** PKCS7 structure
- **cb** Callback function

See: wc_PKCS7_SetOriDecryptCtx

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_SetOriDecryptCb(&pkcs7, myDecryptCallback);
```

```
int wc_PKCS7_AddRecipient_ORI(
    wc_PKCS7 * pkcs7,
    CallbackOriEncrypt cb,
    int options
)
```

Adds ORI recipient.

Parameters:

- **pkcs7** PKCS7 structure
- **cb** Originator encryption callback
- **options** Options flags

See: wc_PKCS7_SetOriDecryptCb

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_AddRecipient_ORI(&pkcs7, myEncryptCallback, 0);
```

```
int wc_PKCS7_SetWrapCEKCb(
    wc_PKCS7 * pkcs7,
    CallbackWrapCEK wrapCEKCb
)
```

Sets CEK wrap callback.

Parameters:

- **pkcs7** PKCS7 structure
- **wrapCEKCb** Wrap CEK callback

See: wc_PKCS7_Init

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_SetWrapCEKCb(&pkcs7, myWrapCEKCallback);
```

```
int wc_PKCS7_SetRsaSignRawDigestCb(
    wc_PKCS7 * pkcs7,
    CallbackRsaSignRawDigest cb
)
```

Sets RSA sign raw digest callback.

Parameters:

- **pkcs7** PKCS7 structure
- **cb** Callback function

See: wc_PKCS7_Init

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_SetRsaSignRawDigestCb(&pkcs7, mySignCallback);
```

```
int wc_PKCS7_EncodeAuthEnvelopedData(
    wc_PKCS7 * pkcs7,
    byte * output,
    word32 outputSz
)
```

Encodes authenticated enveloped data.

Parameters:

- **pkcs7** PKCS7 structure
- **output** Output buffer
- **outputSz** Output buffer size

See: wc_PKCS7_DecodeAuthEnvelopedData

Return:

- Size of encoded data on success
- negative on error

Example

```
int ret = wc_PKCS7_EncodeAuthEnvelopedData(&pkcs7, out, outSz);
```

```
int wc_PKCS7_DecodeAuthEnvelopedData(
    wc_PKCS7 * pkcs7,
    byte * pkiMsg,
    word32 pkiMsgSz,
    byte * output,
    word32 outputSz
)
```

Decodes authenticated enveloped data.

Parameters:

- **pkcs7** PKCS7 structure
- **pkiMsg** Input message
- **pkiMsgSz** Input message size
- **output** Output buffer
- **outputSz** Output buffer size

See: wc_PKCS7_EncodeAuthEnvelopedData

Return:

- Size of decoded data on success
- negative on error

Example

```
int ret = wc_PKCS7_DecodeAuthEnvelopedData(&pkcs7, msg, msgSz, out,
                                           outSz);
```

```
int wc_PKCS7_EncodeEncryptedData(
    wc_PKCS7 * pkcs7,
    byte * output,
    word32 outputSz
)
```

Encodes encrypted data.

Parameters:

- **pkcs7** PKCS7 structure
- **output** Output buffer
- **outputSz** Output buffer size

See: wc_PKCS7_DecodeEncryptedData

Return:

- Size of encoded data on success
- negative on error

Example

```
int ret = wc_PKCS7_EncodeEncryptedData(&pkcs7, out, outSz);
```

```
int wc_PKCS7_SetDecodeEncryptedCb(
    wc_PKCS7 * pkcs7,
    CallbackDecryptContent decryptionCb
)
```

Sets decode encrypted callback.

Parameters:

- **pkcs7** PKCS7 structure
- **decryptionCb** Decryption callback

See: wc_PKCS7_SetDecodeEncryptedCtx

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_SetDecodeEncryptedCb(&pkcs7, myDecryptCallback);
```

```
int wc_PKCS7_SetDecodeEncryptedCtx(
    wc_PKCS7 * pkcs7,
    void * ctx
)
```

Sets decode encrypted context.

Parameters:

- **pkcs7** PKCS7 structure
- **ctx** Context pointer

See: wc_PKCS7_SetDecodeEncryptedCb

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_SetDecodeEncryptedCtx(&pkcs7, myContext);
```

```
int wc_PKCS7_SetStreamMode(
    wc_PKCS7 * pkcs7,
    byte flag,
    CallbackGetContent getContentCb,
    CallbackStreamOut streamOutCb,
    void * ctx
)
```

Sets stream mode for PKCS7.

Parameters:

- **pkcs7** PKCS7 structure
- **flag** Stream mode flag

- **getContentCb** Get content callback
- **streamOutCb** Stream output callback
- **ctx** Context pointer

See: wc_PKCS7_GetStreamMode

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_SetStreamMode(&pkcs7, 1, getContent, streamOut,
                                 ctx);
```

```
int wc_PKCS7_GetStreamMode(
    wc_PKCS7 * pkcs7
)
```

Gets stream mode setting.

Parameters:

- **pkcs7** PKCS7 structure

See: wc_PKCS7_SetStreamMode

Return: Stream mode flag

Example

```
int mode = wc_PKCS7_GetStreamMode(&pkcs7);
```

```
int wc_PKCS7_SetNoCerts(
    wc_PKCS7 * pkcs7,
    byte flag
)
```

Sets no certificates flag.

Parameters:

- **pkcs7** PKCS7 structure
- **flag** No certificates flag

See: wc_PKCS7_GetNoCerts

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_SetNoCerts(&pkcs7, 1);
```

```
int wc_PKCS7_GetNoCerts(
    wc_PKCS7 * pkcs7
)
```

Gets no certificates flag.

Parameters:

- **pkcs7** PKCS7 structure

See: wc_PKCS7_SetNoCerts

Return: No certificates flag

Example

```
int noCerts = wc_PKCS7_GetNoCerts(&pkcs7);
```

```
int wc_PKCS7_EncodeCompressedData(
    wc_PKCS7 * pkcs7,
    byte * output,
    word32 outputSz
)
```

Encodes compressed data.

Parameters:

- **pkcs7** PKCS7 structure
- **output** Output buffer
- **outputSz** Output buffer size

See: wc_PKCS7_DecodeCompressedData

Return:

- Size of encoded data on success
- negative on error

Example

```
int ret = wc_PKCS7_EncodeCompressedData(&pkcs7, out, outSz);
```

```
int wc_PKCS7_DecodeCompressedData(
    wc_PKCS7 * pkcs7,
    byte * pkiMsg,
    word32 pkiMsgSz,
    byte * output,
    word32 outputSz
)
```

Decodes compressed data.

Parameters:

- **pkcs7** PKCS7 structure
- **pkiMsg** Input message
- **pkiMsgSz** Input message size
- **output** Output buffer
- **outputSz** Output buffer size

See: wc_PKCS7_EncodeCompressedData

Return:

- Size of decoded data on success
- negative on error

Example

```
int ret = wc_PKCS7_DecodeCompressedData(&pkcs7, msg, msgSz, out,
                                        outSz);
```

B.43 Algorithms - PKCS11

B.42.4.76 function wc_PKCS7_DecodeCompressedData

B.43.1 Functions

	Name
int	**wc_Pkcs11_Initialize**(Pkcs11Dev * dev, const char * library, void * heap)
void	**wc_Pkcs11_Finalize**(Pkcs11Dev * dev)
int	**wc_Pkcs11Token_Init**(Pkcs11Token * token, Pkcs11Dev * dev, int slotId, const char * tokenName, const unsigned char * userPin, int userPinSz)
void	**wc_Pkcs11Token_Final**(Pkcs11Token * token)
int	**wc_Pkcs11Token_Open**(Pkcs11Token * token, int readWrite)

	Name
void	**wc_Pkcs11Token_Close**(Pkcs11Token * token)
int	**wc_Pkcs11StoreKey**(Pkcs11Token * token, int type, int clear, void * key)
int	**wc_Pkcs11_CryptoDevCb**(int devId, wc_CryptoInfo * info, void * ctx)

B.43.2 Functions Documentation

```
int wc_Pkcs11_Initialize(
    Pkcs11Dev * dev,
    const char * library,
    void * heap
)
```

```
void wc_Pkcs11_Finalize(
    Pkcs11Dev * dev
)
```

```
int wc_Pkcs11Token_Init(
    Pkcs11Token * token,
    Pkcs11Dev * dev,
    int slotId,
    const char * tokenName,
    const unsigned char * userPin,
    int userPinSz
)
```

```
void wc_Pkcs11Token_Final(
    Pkcs11Token * token
)
```

```
int wc_Pkcs11Token_Open(
    Pkcs11Token * token,
    int readWrite
)
```

```
void wc_Pkcs11Token_Close(
    Pkcs11Token * token
)
```

```
int wc_Pkcs11StoreKey(
    Pkcs11Token * token,
    int type,
    int clear,
    void * key
)



int wc_Pkcs11_CryptoDevCb(
    int devId,
    wc_CryptoInfo * info,
    void * ctx
)
```

B.44 Algorithms - PKCS12

B.43.2.8 function wc_Pkcs11_CryptoDevCb More...

B.44.1 Functions

	Name
WC_PKCS12 *	**wc_PKCS12_new**(void)This function creates a new empty WC_PKCS12 structure, using the default heap hint for dynamic memory. It is equivalent to calling wc_PKCS12_new_ex() with a NULL heap. The returned structure must be freed with wc_PKCS12_free().
WC_PKCS12 *	**wc_PKCS12_new_ex**(void * heap)This function creates a new empty WC_PKCS12 structure, associating it with the heap hint given. The heap hint is stored in the structure and is used for every subsequent dynamic allocation and free made on behalf of this WC_PKCS12 object. The returned structure must be freed with wc_PKCS12_free().
void	**wc_PKCS12_free**(WC_PKCS12 * pkcs12)This function frees a WC_PKCS12 structure and all of the memory associated with it, including the parsed authenticated safe and the MAC data. Passing NULL is safe and does nothing.
int	**wc_d2i_PKCS12**(const byte * der, word32 derSz, WC_PKCS12 * pkcs12)This function converts a DER_encoded PKCS #12 (PFX) buffer into a WC_PKCS12 structure. The raw contents of each ContentInfo are stored in the structure without being completely parsed or decoded; call wc_PKCS12_parse() afterwards to decrypt the bundle and recover the private key and certificates.

	Name
int	**wc_d2i_PKCS12_fp**(const char * file, WC_PKCS12 ** pkcs12)This function reads a DER_encoded PKCS #12 (PFX) file from the file system and decodes it into a WC_PKCS12 structure. If *pkcs12 is NULL, a new WC_PKCS12 structure is allocated for the caller and returned through the pkcs12 argument; that structure is freed automatically if the decode fails. In either case, a successfully returned structure must be freed by the caller with wc_PKCS12_free(). This function is not available when NO_FILESYSTEM is defined.
int	**wc_i2d_PKCS12**(WC_PKCS12 * pkcs12, byte ** der, int * derSz)This function encodes a WC_PKCS12 structure into a DER_encoded PKCS #12 (PFX) buffer. It supports three modes of operation, selected by the der and derSz arguments. Passing NULL for der queries the required buffer size only: the size is stored in *derSz, LENGTH_ONLY_E is returned, and no data is written. Passing a non_NULL der whose *der is NULL allocates a buffer of the required size and stores its address in *der, which the caller must free with XFREE() using DYNAMIC_TYPE_PKCS. Passing a non-NULL der whose *der points to a caller-supplied buffer writes the DER into that buffer, and returns BUFFER_E if a non-NULL derSz indicates that the buffer is too small. In that last case *der is advanced on success to one byte past the end of the encoded DER, following the usual i2d convention, so the caller must keep its own copy of the original pointer.

	Name
int	**wc_PKCS12_parse**(WC_PKCS12 * pkcs12, const char * psw, byte ** pkey, word32 * pkeySz, byte ** cert, word32 * certSz, WC_DerCertList ** ca)This function parses and decodes a WC_PKCS12 structure that was previously populated by wc_d2i_PKCS12() or wc_d2i_PKCS12_fp(). The MAC on the bundle is verified with the given password, the contents are decrypted, and the private key, the end entity certificate, and optionally the CA certificate chain are returned to the caller. The key and certificate are returned in newly allocated buffers which the caller is responsible for freeing: use XFREE(pkey, heap, DYNAMIC_TYPE_PUBLIC_KEY) for the key and XFREE(cert, heap, DYNAMIC_TYPE_PKCS) for the certificate, where heap is the heap hint associated with the WC_PKCS12 structure. The CA list, when requested, must be freed with wc_FreeCertList(). The private key is returned with the PKCS #8 header removed; use wc_PKCS12_parse_ex() if the PKCS #8 header should be kept.
int	**wc_PKCS12_parse_ex**(WC_PKCS12 * pkcs12, const char * psw, byte ** pkey, word32 * pkeySz, byte ** cert, word32 * certSz, WC_DerCertList ** ca, int keepKeyHeader)This function behaves exactly like wc_PKCS12_parse() but adds the keepKeyHeader argument, which controls whether the PKCS #8 header is left on the returned private key. Calling wc_PKCS12_parse() is equivalent to calling this function with keepKeyHeader set to 0. The same ownership rules as wc_PKCS12_parse() apply: free the key with XFREE(pkey, heap, DYNAMIC_TYPE_PUBLIC_KEY), the certificate with XFREE(cert, heap, DYNAMIC_TYPE_PKCS), and the CA list with wc_FreeCertList().

	Name
WC_PKCS12 *	**wc_PKCS12_create**(char * pass, word32 passSz, char * name, byte * key, word32 keySz, byte * cert, word32 certSz, WC_DerCertList * ca, int nidKey, int nidCert, int iter, int macIter, int keyType, void * heap)This function creates a new WC_PKCS12 structure from a DER_encoded private key, a DER_encoded certificate, and an optional list of extra certificates. The key and certificate are each placed in their own ContentInfo and optionally encrypted with the password given. A MAC is computed over the result only when macIter is greater than zero. The returned structure can be encoded to DER with wc_i2d_PKCS12() and must be freed with wc_PKCS12_free(). The nidKey and nidCert arguments select the password-based encryption applied to the key and to the certificate respectively. Supported values are PBE_SHA1_RC4_128, PBE_SHA1_DES, PBE_SHA1_DES3, PBE_AES128_CBC and PBE_AES256_CBC. Passing -1 stores the corresponding content unencrypted.
void	**wc_FreeCertList**(WC_DerCertList * list, void * heap)This function frees a WC_DerCertList linked list, including the DER buffer held by each node. It is used to release the CA certificate list returned by wc_PKCS12_parse() and wc_PKCS12_parse_ex(). The heap hint given must match the one associated with the WC_PKCS12 structure the list came from. Passing NULL for the list is safe and does nothing.

B.44.2 Detailed Description

PKCS #12 (RFC 7292) defines the PFX bundle format, a single password protected file that carries a private key together with its certificate and any CA certificates in the chain. It is the format behind the common .p12 and .pfx files. wolfCrypt support is enabled with –enable-pkcs12 or by defining HAVE_PKCS12, and also requires password based key derivation (–enable-pwdbased).

A bundle is read by allocating a WC_PKCS12 structure (wc_PKCS12_new() or wc_PKCS12_new_ex()), decoding the DER into it (wc_d2i_PKCS12() or wc_d2i_PKCS12_fp()), then verifying the MAC and decrypting the contents with wc_PKCS12_parse(), which returns the DER private key, the certificate, and optionally the CA chain.

A bundle is written by building the structure with wc_PKCS12_create() and encoding it with wc_i2d_PKCS12().

See <wolfssl/wolfcrypt/pkcs12.h>.

B.44.3 Functions Documentation

```
WC_PKCS12 * wc_PKCS12_new(
```

```
    void
)
```

This function creates a new empty WC_PKCS12 structure, using the default heap hint for dynamic memory. It is equivalent to calling wc_PKCS12_new_ex() with a NULL heap. The returned structure must be freed with wc_PKCS12_free().

Parameters:

- **none** No parameters.

See:

- wc_PKCS12_new_ex
- wc_PKCS12_free

Return:

- pointer Returns a pointer to a newly allocated WC_PKCS12 structure on success
- NULL Returned if the allocation fails

Example

```
WC_PKCS12* pkcs12 = wc_PKCS12_new();
if (pkcs12 == NULL) {
    // error allocating PKCS12 structure
}

// use the PKCS12 structure

wc_PKCS12_free(pkcs12);
```

```
WC_PKCS12 * wc_PKCS12_new_ex(
    void * heap
)
```

This function creates a new empty WC_PKCS12 structure, associating it with the heap hint given. The heap hint is stored in the structure and is used for every subsequent dynamic allocation and free made on behalf of this WC_PKCS12 object. The returned structure must be freed with wc_PKCS12_free().

Parameters:

- **heap** pointer to a heap hint used for dynamic memory allocation, or NULL to use the default

See:

- wc_PKCS12_new
- wc_PKCS12_free

Return:

- pointer Returns a pointer to a newly allocated WC_PKCS12 structure on success
- NULL Returned if the allocation fails

Example

```
void* heap = NULL; // or a custom static memory heap hint
WC_PKCS12* pkcs12 = wc_PKCS12_new_ex(heap);
if (pkcs12 == NULL) {
    // error allocating PKCS12 structure
}

// use the PKCS12 structure

wc_PKCS12_free(pkcs12);
```

```
void wc_PKCS12_free(
    WC_PKCS12 * pkcs12
)
```

This function frees a WC_PKCS12 structure and all of the memory associated with it, including the parsed authenticated safe and the MAC data. Passing NULL is safe and does nothing.

Parameters:

- **pkcs12** pointer to the WC_PKCS12 structure to free

See:

- wc_PKCS12_new
- wc_PKCS12_new_ex

Return: none No returns.

Example

```
WC_PKCS12* pkcs12 = wc_PKCS12_new();

// initialize and use the PKCS12 structure

wc_PKCS12_free(pkcs12);
```

```
int wc_d2i_PKCS12(
    const byte * der,
    word32 derSz,
    WC_PKCS12 * pkcs12
)
```

This function converts a DER-encoded PKCS #12 (PFX) buffer into a WC_PKCS12 structure. The raw contents of each ContentInfo are stored in the structure without being completely parsed or decoded; call wc_PKCS12_parse() afterwards to decrypt the bundle and recover the private key and certificates.

Parameters:

- **der** pointer to a buffer holding the DER-encoded PKCS #12 bundle
- **derSz** size of the DER buffer
- **pkcs12** pointer to an allocated WC_PKCS12 structure in which to store the decoded bundle

See:

- wc_d2i_PKCS12_fp
- wc_i2d_PKCS12
- wc_PKCS12_parse

Return:

- 0 Returned on successfully decoding the PKCS #12 buffer
- BAD_FUNC_ARG Returned if der or pkcs12 is NULL
- ASN_PARSE_E Returned if there is an error parsing the PKCS #12 structure
- ASN_VERSION_E Returned if the version in the bundle is not supported
- MEMORY_E Returned if there is an error allocating memory

Example

```
WC_PKCS12* pkcs12 = NULL;
byte der[] = { }; // initialize with a DER-encoded PKCS #12 bundle
word32 derSz = sizeof(der);

pkcs12 = wc_PKCS12_new();
if (pkcs12 == NULL) {
    // error allocating PKCS12 structure
}

if (wc_d2i_PKCS12(der, derSz, pkcs12) != 0) {
    // error decoding PKCS12 bundle
}

wc_PKCS12_free(pkcs12);
```

```
int wc_d2i_PKCS12_fp(
    const char * file,
    WC_PKCS12 ** pkcs12
)
```

This function reads a DER-encoded PKCS #12 (PFX) file from the file system and decodes it into a WC_PKCS12 structure. If *pkcs12 is NULL, a new WC_PKCS12 structure is allocated for the caller and returned through the pkcs12 argument; that structure is freed automatically if the decode fails. In

either case, a successfully returned structure must be freed by the caller with wc_PKCS12_free(). This function is not available when NO_FILESYSTEM is defined.

Parameters:

- **file** path of the DER-encoded PKCS #12 file to read
- **pkcs12** pointer to a WC_PKCS12 pointer. If *pkcs12 is NULL, a new structure is allocated and returned here; otherwise the existing structure is used.

See:

- wc_d2i_PKCS12
- wc_i2d_PKCS12
- wc_PKCS12_parse

Return:

- 0 Returned on successfully reading and decoding the file
- BAD_FUNC_ARG Returned if pkcs12 is NULL
- MEMORY_E Returned if there is an error allocating memory
- BAD_PATH_ERROR Returned if the file cannot be opened or read

Example

```
WC_PKCS12* pkcs12 = NULL;

// let wc_d2i_PKCS12_fp allocate the structure for us
if (wc_d2i_PKCS12_fp("./certs/test-servercert.p12", &pkcs12) != 0) {
    // error reading or decoding the PKCS12 file
}

wc_PKCS12_free(pkcs12);
```

```
int wc_i2d_PKCS12(
    WC_PKCS12 * pkcs12,
    byte ** der,
    int * derSz
)
```

This function encodes a WC_PKCS12 structure into a DER-encoded PKCS #12 (PFX) buffer. It supports three modes of operation, selected by the der and derSz arguments. Passing NULL for der queries the required buffer size only: the size is stored in *derSz, LENGTH_ONLY_E is returned, and no data is written. Passing a non-NULL der whose *der is NULL allocates a buffer of the required size and stores its address in *der, which the caller must free with XFREE() using DYNAMIC_TYPE_PKCS. Passing a non-NULL der whose *der points to a caller-supplied buffer writes the DER into that buffer, and returns BUFFER_E if a non-NULL derSz indicates that the buffer is too small. In that last case *der is advanced on success to one byte past the end of the encoded DER, following the usual i2d convention, so the caller must keep its own copy of the original pointer.

Parameters:

- **pkcs12** pointer to the WC_PKCS12 structure to encode
- **der** pointer to a buffer pointer in which to store the encoding, or NULL to query the required size
- **derSz** on a size query, receives the required size. When a caller-supplied buffer is used, holds the size of that buffer on input; a NULL value disables the buffer size check.

See:

- wc_d2i_PKCS12
- wc_PKCS12_create

Return:

- Success On success, returns the size of the DER encoding in bytes
- LENGTH_ONLY_E Returned when der is NULL, indicating that only the required size was computed and stored in *derSz
- BAD_FUNC_ARG Returned if pkcs12 is NULL, if the structure holds no authenticated safe, or if both der and derSz are NULL
- BUFFER_E Returned if a caller-supplied buffer is too small to hold the encoding
- MEMORY_E Returned if there is an error allocating memory

Example

```
WC_PKCS12* pkcs12 = NULL;
byte* der = NULL;
int derSz = 0;

// pkcs12 previously created with wc_PKCS12_create or decoded with
// wc_d2i_PKCS12

// query the required size
if (wc_i2d_PKCS12(pkcs12, NULL, &derSz) != LENGTH_ONLY_E) {
    // error getting the encoded size
}

// let wc_i2d_PKCS12 allocate the buffer for us
if (wc_i2d_PKCS12(pkcs12, &der, NULL) <= 0) {
    // error encoding the PKCS12 bundle
}

XFREE(der, NULL, DYNAMIC_TYPE_PKCS);
wc_PKCS12_free(pkcs12);
```

```
int wc_PKCS12_parse(
    WC_PKCS12 * pkcs12,
    const char * psw,
    byte ** pkey,
    word32 * pkeySz,
    byte ** cert,
    word32 * certSz,
```

```
    WC_DerCertList ** ca
)
```

This function parses and decodes a WC_PKCS12 structure that was previously populated by wc_d2i_PKCS12() or wc_d2i_PKCS12_fp(). The MAC on the bundle is verified with the given password, the contents are decrypted, and the private key, the end entity certificate, and optionally the CA certificate chain are returned to the caller. The key and certificate are returned in newly allocated buffers which the caller is responsible for freeing: use XFREE(pkey, heap, DYNAMIC_TYPE_PUBLIC_KEY) for the key and XFREE(cert, heap, DYNAMIC_TYPE_PKCS) for the certificate, where heap is the heap hint associated with the WC_PKCS12 structure. The CA list, when requested, must be freed with wc_FreeCertList(). The private key is returned with the PKCS #8 header removed; use wc_PKCS12_parse_ex() if the PKCS #8 header should be kept.

Parameters:

- **pkcs12** pointer to a WC_PKCS12 structure holding a decoded bundle
- **psw** NULL-terminated password used to verify the MAC and decrypt the bundle
- **pkey** receives a newly allocated buffer holding the DER-encoded private key
- **pkeySz** receives the size of the private key buffer
- **cert** receives a newly allocated buffer holding the DER-encoded certificate
- **certSz** receives the size of the certificate buffer
- **ca** optional. If non-NULL, receives a linked list of the remaining DER-encoded certificates found in the bundle. Free with wc_FreeCertList().

See:

- wc_PKCS12_parse_ex
- wc_d2i_PKCS12
- wc_FreeCertList

Return:

- 0 Returned on successfully parsing the PKCS #12 bundle
- BAD_FUNC_ARG Returned if pkcs12, psw, pkey, pkeySz, cert or certSz is NULL
- MAC_CMP_FAILED_E Returned if the MAC verification fails, which usually means the password is incorrect
- ASN_PARSE_E Returned if there is an error parsing the contents
- MEMORY_E Returned if there is an error allocating memory
- UNICODE_SIZE_E Returned if the password cannot be converted to the Unicode form required for key derivation

Note: When USER_RSA is enabled this function may return a certificate that is not the pair of the returned key when RSA key pairs are used.

Example

```
WC_PKCS12* pkcs12 = NULL;
byte* key = NULL;
byte* cert = NULL;
WC_DerCertList* ca = NULL;
word32 keySz = 0;
word32 certSz = 0;
```

```
if (wc_d2i_PKCS12_fp("./certs/test-servercert.p12", &pkcs12) != 0) {
    // error reading the PKCS12 file
}

if (wc_PKCS12_parse(pkcs12, "password", &key, &keySz, &cert, &certSz,
                    &ca) != 0) {
    // error parsing the bundle, e.g. wrong password
}

// use the key, cert and ca chain

XFREE(key, NULL, DYNAMIC_TYPE_PUBLIC_KEY);
XFREE(cert, NULL, DYNAMIC_TYPE_PKCS);
wc_FreeCertList(ca, NULL);
wc_PKCS12_free(pkcs12);
```

```
int wc_PKCS12_parse_ex(
    WC_PKCS12 * pkcs12,
    const char * psw,
    byte ** pkey,
    word32 * pkeySz,
    byte ** cert,
    word32 * certSz,
    WC_DerCertList ** ca,
    int keepKeyHeader
)
```

This function behaves exactly like wc_PKCS12_parse() but adds the keepKeyHeader argument, which controls whether the PKCS #8 header is left on the returned private key. Calling wc_PKCS12_parse() is equivalent to calling this function with keepKeyHeader set to 0. The same ownership rules as wc_PKCS12_parse() apply: free the key with XFREE(pkey, heap, DYNAMIC_TYPE_PUBLIC_KEY), the certificate with XFREE(cert, heap, DYNAMIC_TYPE_PKCS), and the CA list with wc_FreeCertList().

Parameters:

- **pkcs12** pointer to a WC_PKCS12 structure holding a decoded bundle
- **psw** NULL-terminated password used to verify the MAC and decrypt the bundle
- **pkey** receives a newly allocated buffer holding the DER-encoded private key
- **pkeySz** receives the size of the private key buffer
- **cert** receives a newly allocated buffer holding the DER-encoded certificate
- **certSz** receives the size of the certificate buffer
- **ca** optional. If non-NULL, receives a linked list of the remaining DER-encoded certificates found in the bundle. Free with wc_FreeCertList().
- **keepKeyHeader** 0 to strip the PKCS #8 header from the returned key, any other value to keep it

See:

- wc_PKCS12_parse
- wc_d2i_PKCS12
- wc_FreeCertList

Return:

- 0 Returned on successfully parsing the PKCS #12 bundle
- BAD_FUNC_ARG Returned if pkcs12, psw, pkey, pkeySz, cert or certSz is NULL
- MAC_CMP_FAILED_E Returned if the MAC verification fails, which usually means the password is incorrect
- ASN_PARSE_E Returned if there is an error parsing the contents
- MEMORY_E Returned if there is an error allocating memory
- UNICODE_SIZE_E Returned if the password cannot be converted to the Unicode form required for key derivation

Note: When USER_RSA is enabled this function may return a certificate that is not the pair of the returned key when RSA key pairs are used.

Example

```
WC_PKCS12* pkcs12 = NULL;
byte* key = NULL;
byte* cert = NULL;
WC_DerCertList* ca = NULL;
word32 keySz = 0;
word32 certSz = 0;

// pkcs12 previously decoded with wc_d2i_PKCS12 or wc_d2i_PKCS12_fp

// keep the PKCS #8 header on the returned private key
if (wc_PKCS12_parse_ex(pkcs12, "password", &key, &keySz, &cert, &certSz,
                       &ca, 1) != 0) {
    // error parsing the bundle
}

XFREE(key, NULL, DYNAMIC_TYPE_PUBLIC_KEY);
XFREE(cert, NULL, DYNAMIC_TYPE_PKCS);
wc_FreeCertList(ca, NULL);
wc_PKCS12_free(pkcs12);
```

```
WC_PKCS12 * wc_PKCS12_create(
    char * pass,
    word32 passSz,
    char * name,
    byte * key,
    word32 keySz,
    byte * cert,
    word32 certSz,
    WC_DerCertList * ca,
    int nidKey,
    int nidCert,
    int iter,
    int macIter,
    int keyType,
    void * heap
)
```

This function creates a new WC_PKCS12 structure from a DER-encoded private key, a DER-encoded certificate, and an optional list of extra certificates. The key and certificate are each placed in their own ContentInfo and optionally encrypted with the password given. A MAC is computed over the result only when macIter is greater than zero. The returned structure can be encoded to DER with wc_i2d_PKCS12() and must be freed with wc_PKCS12_free(). The nidKey and nidCert arguments select the password-based encryption applied to the key and to the certificate respectively. Supported values are PBE_SHA1_RC4_128, PBE_SHA1_DES, PBE_SHA1_DES3, PBE_AES128_CBC and PBE_AES256_CBC. Passing -1 stores the corresponding content unencrypted.

Parameters:

- **pass** password to use for encryption and for the MAC
- **passSz** size of the password buffer
- **name** friendlyName to use. Not currently used.
- **key** buffer holding the DER-encoded private key
- **keySz** size of the key buffer
- **cert** buffer holding the DER-encoded certificate
- **certSz** size of the certificate buffer
- **ca** optional linked list of additional DER-encoded certificates to include, or NULL
- **nidKey** encryption to apply to the private key, or -1 for none
- **nidCert** encryption to apply to the certificate, or -1 for none
- **iter** number of iterations to use for the encryption. Values of 0 or less select WC_PKCS12_ITT_DEFAULT.
- **macIter** number of iterations to use when creating the MAC. A MAC is only created when this is greater than zero. Unlike iter, a value of zero or less does not select a default: no MAC is created at all, and the resulting bundle has no MAC integrity protection.
- **keyType** flag for a signature and/or encryption key. Not currently used.
- **heap** pointer to a heap hint used for dynamic memory allocation, or NULL to use the default

See:

- wc_i2d_PKCS12
- wc_PKCS12_parse
- wc_PKCS12_free

Return:

- pointer Returns a pointer to a newly created WC_PKCS12 structure on success
- NULL Returned if the RNG cannot be initialized, if memory allocation fails, if an unsupported nidKey or nidCert is given, or if the bundle cannot be built

Note: The name and keyType arguments are accepted for API compatibility but are not currently used.

Example

```
WC_PKCS12* pkcs12 = NULL;
byte* der = NULL;
char pass[] = "password";
byte key[] = { };  // initialize with a DER-encoded private key
byte cert[] = { }; // initialize with a DER-encoded certificate

pkcs12 = wc_PKCS12_create(pass, sizeof(pass) - 1, NULL,
                          key, sizeof(key), cert, sizeof(cert), NULL,
                          PBE_AES256_CBC, PBE_AES256_CBC,
```

```
                            WC_PKCS12_ITT_DEFAULT, WC_PKCS12_ITT_DEFAULT,
                            0, NULL);
if (pkcs12 == NULL) {
    // error creating the PKCS12 bundle
}

if (wc_i2d_PKCS12(pkcs12, &der, NULL) <= 0) {
    // error encoding the bundle to DER
}

XFREE(der, NULL, DYNAMIC_TYPE_PKCS);
wc_PKCS12_free(pkcs12);
```

```
void wc_FreeCertList(
    WC_DerCertList * list,
    void * heap
)
```

This function frees a WC_DerCertList linked list, including the DER buffer held by each node. It is used to release the CA certificate list returned by wc_PKCS12_parse() and wc_PKCS12_parse_ex(). The heap hint given must match the one associated with the WC_PKCS12 structure the list came from. Passing NULL for the list is safe and does nothing.

Parameters:

- **list** pointer to the head of the WC_DerCertList to free
- **heap** pointer to the heap hint used when the list was allocated

See:

- wc_PKCS12_parse
- wc_PKCS12_parse_ex

Return: none No returns.

Example

```
WC_DerCertList* ca = NULL;

// ca populated by a previous call to wc_PKCS12_parse

wc_FreeCertList(ca, NULL);
```

B.45 Algorithms - Poly1305

B.44.3.10 function wc_FreeCertList

B.45.1 Functions

	Name
int	**wc_Poly1305SetKey**(Poly1305 * poly1305, const byte * key, word32 kySz)This function sets the key for a Poly1305 context structure, initializing it for hashing. Note: A new key should be set after generating a message hash with wc_Poly1305Final to ensure security.
int	**wc_Poly1305Update**(Poly1305 * poly1305, const byte * m, word32 bytes)This function updates the message to hash with the Poly1305 structure.
int	**wc_Poly1305Final**(Poly1305 * poly1305, byte * tag)This function calculates the hash of the input messages and stores the result in mac. After this is called, the key should be reset.
int	**wc_Poly1305_MAC**(Poly1305 * ctx, const byte * additional, word32 addSz, const byte * input, word32 sz, byte * tag, word32 tagSz)Takes in an initialized Poly1305 struct that has a key loaded and creates a MAC (tag) using recent TLS AEAD padding scheme.
int	**wc_Poly1305_Pad**(Poly1305 * ctx, word32 lenToPad)Adds padding to Poly1305 context.
int	**wc_Poly1305_EncodeSizes**(Poly1305 * ctx, word32 aadSz, word32 dataSz)Encodes AAD and data sizes for Poly1305.
int	**wc_Poly1305_EncodeSizes64**(Poly1305 * ctx, word64 aadSz, word64 dataSz)Encodes AAD and data sizes for Poly1305 using 64-bit values.

B.45.2 Functions Documentation

```
int wc_Poly1305SetKey(
    Poly1305 * poly1305,
    const byte * key,
    word32 kySz
)
```

This function sets the key for a Poly1305 context structure, initializing it for hashing. Note: A new key should be set after generating a message hash with wc_Poly1305Final to ensure security.

Parameters:

- **ctx** pointer to a Poly1305 structure to initialize
- **key** pointer to the buffer containing the key to use for hashing
- **keySz** size of the key in the buffer. Should be 32 bytes

See:

- wc_Poly1305Update

- wc_Poly1305Final

Return:

- 0 Returned on successfully setting the key and initializing the Poly1305 structure
- BAD_FUNC_ARG Returned if the given key is not 32 bytes long, or the Poly1305 context is NULL

Example

```
Poly1305 enc;
byte key[] = { initialize with 32 byte key to use for hashing };
wc_Poly1305SetKey(&enc, key, sizeof(key));
```

```
int wc_Poly1305Update(
    Poly1305 * poly1305,
    const byte * m,
    word32 bytes
)
```

This function updates the message to hash with the Poly1305 structure.

Parameters:

- **ctx** pointer to a Poly1305 structure for which to update the message to hash
- **m** pointer to the buffer containing the message which should be added to the hash
- **bytes** size of the message to hash

See:

- wc_Poly1305SetKey
- wc_Poly1305Final

Return:

- 0 Returned on successfully updating the message to hash
- BAD_FUNC_ARG Returned if the Poly1305 structure is NULL

Example

```
Poly1305 enc;
byte key[] = { }; // initialize with 32 byte key to use for encryption

byte msg[] = { }; // initialize with message to hash
wc_Poly1305SetKey(&enc, key, sizeof(key));

if( wc_Poly1305Update(key, msg, sizeof(msg)) != 0 ) {
    // error updating message to hash
}
```

```
int wc_Poly1305Final(
    Poly1305 * poly1305,
    byte * tag
)
```

This function calculates the hash of the input messages and stores the result in mac. After this is called, the key should be reset.

Parameters:

- **ctx** pointer to a Poly1305 structure with which to generate the MAC
- **mac** pointer to the buffer in which to store the MAC. Should be POLY1305_DIGEST_SIZE (16 bytes) wide

See:

- wc_Poly1305SetKey
- wc_Poly1305Update

Return:

- 0 Returned on successfully computing the final MAC
- BAD_FUNC_ARG Returned if the Poly1305 structure is NULL

Example

```
Poly1305 enc;
byte mac[POLY1305_DIGEST_SIZE]; // space for a 16 byte mac

byte key[] = { }; // initialize with 32 byte key to use for encryption

byte msg[] = { }; // initialize with message to hash
wc_Poly1305SetKey(&enc, key, sizeof(key));
wc_Poly1305Update(key, msg, sizeof(msg));

if ( wc_Poly1305Final(&enc, mac) != 0 ) {
    // error computing final MAC
}
```

```
int wc_Poly1305_MAC(
    Poly1305 * ctx,
    const byte * additional,
    word32 addSz,
    const byte * input,
    word32 sz,
    byte * tag,
    word32 tagSz
)
```

Takes in an initialized Poly1305 struct that has a key loaded and creates a MAC (tag) using recent TLS AEAD padding scheme.

Parameters:

- **ctx** Initialized Poly1305 struct to use
- **additional** Additional data to use
- **addSz** Size of additional buffer
- **input** Input buffer to create tag from
- **sz** Size of input buffer
- **tag** Buffer to hold created tag
- **tagSz** Size of input tag buffer (must be at least WC_POLY1305_MAC_SZ(16))

See:

- wc_Poly1305SetKey
- wc_Poly1305Update
- wcPoly1305Final

Return:

- 0 Success
- BAD_FUNC_ARG Returned if ctx, input, or tag is null or if additional is null and addSz is greater than 0 or if tagSz is less than WC_POLY1305_MAC_SZ.

Example

```
Poly1305 ctx;
byte key[] = { }; // initialize with 32 byte key to use for hashing
byte additional[] = { }; // initialize with additional data
byte msg[] = { }; // initialize with message
byte tag[16];

wc_Poly1305SetKey(&ctx, key, sizeof(key));
if(wc_Poly1305_MAC(&ctx, additional, sizeof(additional), (byte*)msg,
sizeof(msg), tag, sizeof(tag)) != 0)
{
    // Handle the error
}
```

```
int wc_Poly1305_Pad(
    Poly1305 * ctx,
    word32 lenToPad
)
```

Adds padding to Poly1305 context.

Parameters:

- **ctx** Poly1305 context

- **lenToPad** Length to pad

See: wc_Poly1305_MAC

Return:

- 0 on success
- BAD_FUNC_ARG if ctx is NULL

Example

```
Poly1305 ctx;
byte key[32];
wc_Poly1305SetKey(&ctx, key, sizeof(key));
int ret = wc_Poly1305_Pad(&ctx, 10);
```

```
int wc_Poly1305_EncodeSizes(
    Poly1305 * ctx,
    word32 aadSz,
    word32 dataSz
)
```

Encodes AAD and data sizes for Poly1305.

Parameters:

- **ctx** Poly1305 context
- **aadSz** Additional authenticated data size
- **dataSz** Data size

See: wc_Poly1305_MAC

Return:

- 0 on success
- BAD_FUNC_ARG if ctx is NULL

Example

```
Poly1305 ctx;
byte key[32];
wc_Poly1305SetKey(&ctx, key, sizeof(key));
int ret = wc_Poly1305_EncodeSizes(&ctx, 16, 100);
```

```
int wc_Poly1305_EncodeSizes64(
    Poly1305 * ctx,
    word64 aadSz,
    word64 dataSz
)
```

Encodes AAD and data sizes for Poly1305 using 64-bit values.

Parameters:

- **ctx** Poly1305 context
- **aadSz** Additional authenticated data size
- **dataSz** Data size

See: wc_Poly1305_EncodeSizes

Return:

- 0 on success
- BAD_FUNC_ARG if ctx is NULL

Example

```
Poly1305 ctx;
byte key[32];
wc_Poly1305SetKey(&ctx, key, sizeof(key));
int ret = wc_Poly1305_EncodeSizes64(&ctx, 16, 100);
```

B.46 Platform Security Architecture (PSA) API

B.45.2.7 function wc_Poly1305_EncodeSizes64

B.46.1 Functions

	Name
int	**wolfSSL_CTX_psa_enable**(WOLFSSL_CTX * ctx)This function enables PSA support on the given context.
int	**wolfSSL_set_psa_ctx**(WOLFSSL * ssl, struct psa_ssl_ctx * ctx)This function setup the PSA context for the given SSL session.
void	**wolfSSL_free_psa_ctx**(struct psa_ssl_ctx * ctx)This function releases the resources used by a PSA context.
int	**wolfSSL_psa_set_private_key_id**(struct psa_ssl_ctx * ctx, psa_key_id_t id)This function set the private key used by an SSL session.
int	**wc_psa_get_random**(unsigned char * out, word32 sz)This function generates random bytes using the PSA crypto API. This is a wrapper around the PSA random number generation functions.

	Name
int	**wc_psa_aes_encrypt_decrypt**(Aes * aes, const uint8_t * input, uint8_t * output, size_t length, psa_algorithm_t alg, int direction)This function performs AES encryption or decryption using the PSA crypto API. It supports various AES modes through the algorithm parameter.

B.46.2 Functions Documentation

```
int wolfSSL_CTX_psa_enable(
    WOLFSSL_CTX * ctx
)
```

This function enables PSA support on the given context.

Parameters:

- **ctx** pointer to the WOLFSSL_CTX object on which the PSA support must be enabled

See: wolfSSL_set_psa_ctx

Return:

- WOLFSSL_SUCCESS on success
- BAD_FUNC_ARG if ctx == NULL

Example

```
WOLFSSL_CTX *ctx;
ctx = wolfSSL_CTX_new(wolfTLSv1_2_client_method());
if (!ctx)
    return NULL;
ret = wolfSSL_CTX_psa_enable(ctx);
if (ret != WOLFSSL_SUCCESS)
    printf("can't enable PSA on ctx");
```

```
int wolfSSL_set_psa_ctx(
    WOLFSSL * ssl,
    struct psa_ssl_ctx * ctx
)
```

This function setup the PSA context for the given SSL session.

Parameters:

- **ssl** pointer to the WOLFSSL where the ctx will be enabled
- **ctx** pointer to a struct psa_ssl_ctx (must be unique for a ssl session)

See:

- wolfSSL_psa_set_private_key_id
- wolfSSL_psa_free_psa_ctx

Return:

- WOLFSSL_SUCCESS on success
- BAD_FUNC_ARG if ssl or ctx are NULL

This function setup the PSA context for the TLS callbacks to the given SSL session. At the end of the session, the resources used by the context should be freed using wolfSSL_free_psa_ctx().

Example

```
// Create new ssl session
WOLFSSL *ssl;
struct psa_ssl_ctx psa_ctx = { 0 };
ssl = wolfSSL_new(ctx);
if (!ssl)
    return NULL;
// setup PSA context
ret = wolfSSL_set_psa_ctx(ssl, ctx);
```

```
void wolfSSL_free_psa_ctx(
    struct psa_ssl_ctx * ctx
)
```

This function releases the resources used by a PSA context.

Parameters:

- **ctx** pointer to a struct psa_ssl_ctx

See: wolfSSL_set_psa_ctx

```
int wolfSSL_psa_set_private_key_id(
    struct psa_ssl_ctx * ctx,
    psa_key_id_t id
)
```

This function set the private key used by an SSL session.

Parameters:

- **ctx** pointer to a struct psa_ssl_ctx
- **id** PSA id of the key to be used as private key

See: wolfSSL_set_psa_ctx

Example

```
// Create new ssl session
WOLFSSL *ssl;
struct psa_ssl_ctx psa_ctx = { 0 };
psa_key_id_t key_id;

// key provisioning already done
get_private_key_id(&key_id);

ssl = wolfSSL_new(ctx);
if (!ssl)
    return NULL;

wolfSSL_psa_set_private_key_id(&psa_ctx, key_id);
wolfSSL_set_psa_ctx(ssl, ctx);
```

```
int wc_psa_get_random(
    unsigned char * out,
    word32 sz
)
```

This function generates random bytes using the PSA crypto API. This is a wrapper around the PSA random number generation functions.

Parameters:

- **out** pointer to buffer to store random bytes
- **sz** number of random bytes to generate

See: wc_RNG_GenerateBlock

Return:

- 0 On success
- Negative value on error

Example

```
byte random[32];

int ret = wc_psa_get_random(random, sizeof(random));
if (ret != 0) {
    // error generating random bytes
}
```

```
int wc_psa_aes_encrypt_decrypt(
    Aes * aes,
    const uint8_t * input,
    uint8_t * output,
    size_t length,
    psa_algorithm_t alg,
    int direction
)
```

This function performs AES encryption or decryption using the PSA crypto API. It supports various AES modes through the algorithm parameter.

Parameters:

- **aes** pointer to initialized Aes structure
- **input** pointer to input data buffer
- **output** pointer to output data buffer
- **length** length of data to process
- **alg** PSA algorithm identifier specifying the AES mode
- **direction** encryption (1) or decryption (0)

See:

- wc_AesEncrypt
- wc_AesDecrypt

Return:

- 0 On success
- Negative value on error

Example

```
Aes aes;
byte key[16] = { }; // AES key
byte input[16] = { }; // plaintext
byte output[16];

wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesSetKey(&aes, key, sizeof(key), NULL, AES_ENCRYPTION);
int ret = wc_psa_aes_encrypt_decrypt(&aes, input, output,
                                     sizeof(input),
                                     PSA_ALG_ECB_NO_PADDING, 1);
```

B.47 Algorithms - PUF

B.46.2.6 function wc_psa_aes_encrypt_decrypt

B.47.1 Functions

	Name
int	**wc_PufInit**(wc_PufCtx * ctx)Initialize a wc_PufCtx structure, zeroing all fields. Must be called before any other PUF operations.
int	**wc_PufReadSram**(wc_PufCtx * ctx, const byte * sramAddr, word32 sramSz)Read raw SRAM data into the PUF context. The sramAddr should point to a NOLOAD linker section to preserve the power_on state. The required size, WC_PUF_RAW_BYTES, scales with WC_PUF_NUM_CODEWORDS (256 bytes at the default 16 codewords).
int	**wc_PufCheckSram**(const byte * sramAddr, word32 sramSz, word32 * onesCount)Health test a candidate raw SRAM readout without loading it into a context. Rejects a readout that cannot be SRAM power_on noise: any 128_bit block that is all zero or all ones, any block that repeats the block before it, or a total Hamming weight outside the WC_PUF_HW_MIN_PCT to WC_PUF_HW_MAX_PCT band (default 35% to 65% of WC_PUF_RAW_BITS).
int	**wc_PufEnroll**(wc_PufCtx * ctx)Perform PUF enrollment. Encodes raw SRAM using the selected BCH(127,k,t) profile (WC_PUF_BCH_T) and generates public helper data (WC_PUF_HELPER_BYTES). After enrollment the context is ready for key derivation and identity retrieval.
int	**wc_PufReconstruct**(wc_PufCtx * ctx, const byte * helperData, word32 helperSz)Reconstruct stable PUF bits from noisy SRAM using stored helper data. BCH error correction corrects up to WC_PUF_BCH_T bit flips per 127-bit codeword. The helper data and build configuration must match the enrollment that produced them.
int	**wc_PufDeriveKey**(wc_PufCtx * ctx, const byte * info, word32 infoSz, byte * key, word32 keySz)Derive a cryptographic key from PUF stable bits using HKDF. Uses SHA-256 by default, or SHA3-256 when WC_PUF_SHA3 is defined. The info parameter provides domain separation for multiple keys. Requires HAVE_HKDF.
int	**wc_PufGetIdentity**(wc_PufCtx * ctx, byte * id, word32 idSz)Retrieve the device identity hash (SHA_256 or SHA3_256 of stable bits). Deterministic for a given device.

	Name
int	**wc_PufGetParams**(int * m, int * n, int * k, int * t, int * numCodewords)Report the compile_time PUF profile parameters: field size m, codeword length n, message length k, error_correction capability t, and the number of codewords. Each output pointer is optional (may be NULL), but an all_NULL call is treated as a usage error. Enrollment and reconstruction firmware must agree on all of these (and the hash); persist WC_PUF_PROFILE_ID (which also encodes the hash selection) with the helper data and compare before reconstruction to detect a build mismatch.
word32	**wc_PufGetProfileId**(void)Return the profile fingerprint the LIBRARY was compiled with. The WC_PUF_PROFILE_ID macro necessarily reflects the including application's own build, so comparing the two detects a library/application mismatch - including a differing hash selection - that no length check can see.
int	**wc_PufGetHelperData**(wc_PufCtx * ctx, byte * helper, word32 helperSz)Copy out the enrollment helper data. Use this rather than reading wc_PufCtx.helperData directly: the size varies with the selected profile and with WC_PUF_HELPER_COMPACT.
int	**wc_PufReconstructEx**(wc_PufCtx * ctx, const byte * helperData, word32 helperSz, word32 profileId)Reconstruct as wc_PufReconstruct, but first check the caller's stored profile id against the library's. Helper-data size does not depend on t in the default layout, so the length check alone cannot detect helper data enrolled by a differently configured build - which would otherwise decode to a silently wrong key. Persist WC_PUF_PROFILE_ID alongside the helper data at enrollment and pass it back here.
int	**wc_PufZeroize**(wc_PufCtx * ctx)Securely zeroize all sensitive data in the PUF context using ForceZero. Call when PUF is no longer needed.
int	**wc_PufSetTestData**(wc_PufCtx * ctx, const byte * data, word32 sz)Inject synthetic SRAM test data for testing without hardware. Only available when WOLFSSL_PUF_TEST is defined.

B.47.2 Functions Documentation

```
int wc_PufInit(
```

```
    wc_PufCtx * ctx
)
```

Initialize a wc_PufCtx structure, zeroing all fields. Must be called before any other PUF operations.

Parameters:

- **ctx** pointer to wc_PufCtx structure to initialize

See:

- wc_PufReadSram
- wc_PufEnroll
- wc_PufZeroize

Return:

- 0 on success
- BAD_FUNC_ARG if ctx is NULL

The SRAM PUF uses a configurable BCH(127,k,t) fuzzy extractor over GF(2^7) with HKDF key derivation. WC_PUF_BCH_T selects the error-correction strength (t=7, 10 default, 13, or 15) and WC_PUF_NUM_CODEWORDS (default 16) trades SRAM footprint and helper-data size (WC_PUF_HELPER_BYTES) against derived-key entropy. Only the (n - k) parity bits per codeword carry information; the leading k bits of each helper codeword are identically zero, so size OTP/flash accordingly. WC_PUF_HELPER_COMPACT stores only those parity bits, shrinking helper data to 39-72% of the default. It changes the stored format and so is opt-in; the default layout stays compatible with helper data enrolled by wolfSSL 5.9.2.

Enrollment and reconstruction must use identical WC_PUF_BCH_T and WC_PUF_NUM_CODEWORDS (and the same hash); persist WC_PUF_PROFILE_ID with the helper data at enrollment and pass it to wc_PufReconstructEx, or compare it against wc_PufGetProfileId(), to catch a build mismatch.

Every readout handed to wc_PufReadSram() is health tested first (see wc_PufCheckSram). A degenerate readout _ all zero, all ones, or a repeating block _ would otherwise pass cleanly through encoding, masking, decoding and HKDF and derive a key that is the same on every device. WC_PUF_HW_MIN_PCT and WC_PUF_HW_MAX_PCT (default 35 and 65) set the Hamming-weight band the readout must fall inside.

The test rejects degenerate readouts, not every already-written region: ordinary firmware content - .data copied from flash, a string table, a previous boot stage - is identical on every device yet neither constant nor strongly biased, so it can pass. Sampling the region from reset, before .bss/.data initialization and before it is used as stack or heap, remains a requirement of correct NOLOAD placement rather than something this test can enforce.

For a complete bare-metal example (tested on NUCLEO-H563ZI), see https://github.com/wolfSSL/wolfssl-examples/tree/master/puf

Example

```
wc_PufCtx ctx;
ret = wc_PufInit(&ctx);
```

```
int wc_PufReadSram(
    wc_PufCtx * ctx,
    const byte * sramAddr,
    word32 sramSz
)
```

Read raw SRAM data into the PUF context. The sramAddr should point to a NOLOAD linker section to preserve the power-on state. The required size, WC_PUF_RAW_BYTES, scales with WC_PUF_NUM_CODEWORDS (256 bytes at the default 16 codewords).

Parameters:

- **ctx** pointer to wc_PufCtx structure
- **sramAddr** pointer to raw SRAM memory region
- **sramSz** size of SRAM buffer (must be >= WC_PUF_RAW_BYTES)

See:

- wc_PufInit
- wc_PufCheckSram
- wc_PufEnroll
- wc_PufReconstruct

Return:

- 0 on success
- BAD_FUNC_ARG if ctx or sramAddr is NULL
- PUF_READ_E if sramSz < WC_PUF_RAW_BYTES, or if the readout fails the health test

The readout is health tested with wc_PufCheckSram() before it is accepted. A degenerate readout _ typically a region already cleared by .bss init, the common bring_up mistake of sampling after C runtime startup _ is rejected with PUF_READ_E, and the context is left unusable by wc_PufEnroll() and wc_PufReconstruct(). The test catches degenerate shapes, not every already-written region, so it does not remove the requirement to sample from reset.

Example

```
__attribute__((section(".puf_sram")))
static volatile uint8_t puf_sram[WC_PUF_RAW_BYTES];
wc_PufReadSram(&ctx, (const byte*)puf_sram, sizeof(puf_sram));
```

```
int wc_PufCheckSram(
    const byte * sramAddr,
    word32 sramSz,
    word32 * onesCount
)
```

Health test a candidate raw SRAM readout without loading it into a context. Rejects a readout that cannot be SRAM power-on noise: any 128-bit block that is all zero or all ones, any block that repeats the block before it, or a total Hamming weight outside the WC_PUF_HW_MIN_PCT to WC_PUF_HW_MAX_PCT band (default 35% to 65% of WC_PUF_RAW_BITS).

Parameters:

- **sramAddr** pointer to raw SRAM memory region
- **sramSz** size of SRAM buffer (must be >= WC_PUF_RAW_BYTES)
- **onesCount** optional; receives the number of one bits in the first WC_PUF_RAW_BYTES of the readout, out of WC_PUF_RAW_BITS. Written whenever sramAddr is non-NULL and sramSz is large enough, whatever the verdict. It is a property of the raw PUF material, so treat it as a bring-up measurement and do not report it from production firmware

See:

- wc_PufReadSram
- wc_PufEnroll

Return:

- 0 if the readout is plausible PUF material
- BAD_FUNC_ARG if sramAddr is NULL
- PUF_READ_E if sramSz < WC_PUF_RAW_BYTES, or if the readout fails any of the checks

wc_PufReadSram() applies this test to every readout, so calling it directly is only needed to qualify a candidate SRAM region during board bring-up, or to report why a read was refused. onesCount is optional and is written whenever the size check passes - including when the readout is then rejected, so the measured bias of a rejected region is still available. It is left untouched when sramAddr is NULL or sramSz is short.

Example

```
word32 ones = 0;
ret = wc_PufCheckSram((const byte*)puf_sram, sizeof(puf_sram), &ones);
printf("PUF SRAM bias %u/%u ones, ret %d\n",
       (unsigned)ones, (unsigned)WC_PUF_RAW_BITS, ret);
```

```
int wc_PufEnroll(
    wc_PufCtx * ctx
)
```

Perform PUF enrollment. Encodes raw SRAM using the selected BCH(127,k,t) profile (WC_PUF_BCH_T) and generates public helper data (WC_PUF_HELPER_BYTES). After enrollment the context is ready for key derivation and identity retrieval.

Parameters:

- **ctx** pointer to wc_PufCtx (must have SRAM data loaded)

See:

- wc_PufReadSram
- wc_PufReconstruct

- wc_PufDeriveKey

Return:

- 0 on success
- BAD_FUNC_ARG if ctx is NULL
- PUF_ENROLL_E if enrollment fails

Example

```
wc_PufEnroll(&ctx);
XMEMCPY(helperData, ctx.helperData, WC_PUF_HELPER_BYTES);
```

```
int wc_PufReconstruct(
    wc_PufCtx * ctx,
    const byte * helperData,
    word32 helperSz
)
```

Reconstruct stable PUF bits from noisy SRAM using stored helper data. BCH error correction corrects up to WC_PUF_BCH_T bit flips per 127-bit codeword. The helper data and build configuration must match the enrollment that produced them.

Parameters:

- **ctx** pointer to wc_PufCtx (must have SRAM data loaded)
- **helperData** pointer to helper data from previous enrollment
- **helperSz** size of helper data (>= WC_PUF_HELPER_BYTES)

See:

- wc_PufEnroll
- wc_PufDeriveKey
- wc_PufGetIdentity

Return:

- 0 on success
- BAD_FUNC_ARG if ctx or helperData is NULL
- PUF_RECONSTRUCT_E on failure (too many bit errors or helperSz too small)

Example

```
wc_PufReconstruct(&ctx, helperData, sizeof(helperData));
```

```
int wc_PufDeriveKey(
    wc_PufCtx * ctx,
    const byte * info,
    word32 infoSz,
    byte * key,
    word32 keySz
)
```

Derive a cryptographic key from PUF stable bits using HKDF. Uses SHA-256 by default, or SHA3-256 when WC_PUF_SHA3 is defined. The info parameter provides domain separation for multiple keys. Requires HAVE_HKDF.

Parameters:

- **ctx** pointer to wc_PufCtx (must be enrolled or reconstructed)
- **info** optional context info for domain separation (may be NULL; when NULL, infoSz is treated as 0)
- **infoSz** size of info in bytes
- **key** output buffer for derived key
- **keySz** desired key size in bytes

See:

- wc_PufEnroll
- wc_PufReconstruct
- wc_PufGetIdentity

Return:

- 0 on success
- BAD_FUNC_ARG if ctx or key is NULL, or keySz is 0
- PUF_DERIVE_KEY_E if PUF not ready or HKDF fails

Example

```
byte key[32];
const byte info[] = "my-app-key";
wc_PufDeriveKey(&ctx, info, sizeof(info), key, sizeof(key));
```

```
int wc_PufGetIdentity(
    wc_PufCtx * ctx,
    byte * id,
    word32 idSz
)
```

Retrieve the device identity hash (SHA-256 or SHA3-256 of stable bits). Deterministic for a given device.

Parameters:

- **ctx** pointer to wc_PufCtx (must be enrolled or reconstructed)

- **id** output buffer for identity hash
- **idSz** size of id buffer (>= WC_PUF_ID_SZ, 32 bytes)

See:

- wc_PufEnroll
- wc_PufReconstruct
- wc_PufDeriveKey

Return:

- 0 on success
- BAD_FUNC_ARG if ctx or id is NULL
- PUF_IDENTITY_E if PUF not ready or idSz < WC_PUF_ID_SZ

Example

```
byte identity[WC_PUF_ID_SZ];
wc_PufGetIdentity(&ctx, identity, sizeof(identity));
```

```
int wc_PufGetParams(
    int * m,
    int * n,
    int * k,
    int * t,
    int * numCodewords
)
```

Report the compile-time PUF profile parameters: field size m, codeword length n, message length k, error-correction capability t, and the number of codewords. Each output pointer is optional (may be NULL), but an all-NULL call is treated as a usage error. Enrollment and reconstruction firmware must agree on all of these (and the hash); persist WC_PUF_PROFILE_ID (which also encodes the hash selection) with the helper data and compare before reconstruction to detect a build mismatch.

Parameters:

- **m** optional output for the GF field exponent (7 for GF(2^7))
- **n** optional output for the codeword length (127)
- **k** optional output for the message length (per WC_PUF_BCH_T)
- **t** optional output for the error-correction capability (WC_PUF_BCH_T)
- **numCodewords** optional output for WC_PUF_NUM_CODEWORDS

See:

- wc_PufEnroll
- wc_PufReconstruct

Return:

- 0 on success
- BAD_FUNC_ARG if every output pointer is NULL

Example

```
int t, numCodewords;
wc_PufGetParams(NULL, NULL, NULL, &t, &numCodewords);
```

```
word32 wc_PufGetProfileId(
    void
)
```

Return the profile fingerprint the LIBRARY was compiled with. The WC_PUF_PROFILE_ID macro necessarily reflects the including application's own build, so comparing the two detects a library/application mismatch - including a differing hash selection - that no length check can see.

See:

- wc_PufGetParams
- wc_PufReconstructEx

Return: the library's WC_PUF_PROFILE_ID

Example

```
if (wc_PufGetProfileId() != WC_PUF_PROFILE_ID) {
    /* library and application were built with different PUF settings */
}
```

```
int wc_PufGetHelperData(
    wc_PufCtx * ctx,
    byte * helper,
    word32 helperSz
)
```

Copy out the enrollment helper data. Use this rather than reading wc_PufCtx.helperData directly: the size varies with the selected profile and with WC_PUF_HELPER_COMPACT.

Parameters:

- **ctx** pointer to an enrolled wc_PufCtx
- **helper** output buffer for the helper data
- **helperSz** size of helper in bytes (>= WC_PUF_HELPER_BYTES)

See:

- wc_PufEnroll
- wc_PufReconstructEx

Return:

- 0 on success
- BAD_FUNC_ARG if ctx or helper is NULL
- PUF_ENROLL_E if the context is not enrolled, or helperSz is less than WC_PUF_HELPER_BYTES

Example

```
byte helper[WC_PUF_HELPER_BYTES];
wc_PufGetHelperData(&ctx, helper, sizeof(helper));
```

```
int wc_PufReconstructEx(
    wc_PufCtx * ctx,
    const byte * helperData,
    word32 helperSz,
    word32 profileId
)
```

Reconstruct as wc_PufReconstruct, but first check the caller's stored profile id against the library's. Helper-data size does not depend on t in the default layout, so the length check alone cannot detect helper data enrolled by a differently configured build - which would otherwise decode to a silently wrong key. Persist WC_PUF_PROFILE_ID alongside the helper data at enrollment and pass it back here.

Parameters:

- **ctx** pointer to wc_PufCtx (must have SRAM data loaded)
- **helperData** pointer to helper data from a previous enrollment
- **helperSz** size of helper data (>= WC_PUF_HELPER_BYTES)
- **profileId** the WC_PUF_PROFILE_ID recorded at enrollment

See:

- wc_PufReconstruct
- wc_PufGetProfileId

Return:

- 0 on success
- BAD_FUNC_ARG if ctx or helperData is NULL, or profileId does not match the library's profile
- PUF_RECONSTRUCT_E on failure (too many bit errors or helperSz too small)

Example

```
wc_PufReconstructEx(&ctx, helper, sizeof(helper), savedProfileId);
```

```
int wc_PufZeroize(
    wc_PufCtx * ctx
)
```

Securely zeroize all sensitive data in the PUF context using ForceZero. Call when PUF is no longer needed.

Parameters:

- **ctx** pointer to wc_PufCtx to zeroize

See: wc_PufInit

Return:

- 0 on success
- BAD_FUNC_ARG if ctx is NULL

Example

```
wc_PufZeroize(&ctx);
```

```
int wc_PufSetTestData(
    wc_PufCtx * ctx,
    const byte * data,
    word32 sz
)
```

Inject synthetic SRAM test data for testing without hardware. Only available when WOLFSSL_PUF_TEST is defined.

Parameters:

- **ctx** pointer to wc_PufCtx
- **data** pointer to synthetic SRAM data
- **sz** size of data (>= WC_PUF_RAW_BYTES)

See:

- wc_PufInit
- wc_PufReadSram

Return:

- 0 on success
- BAD_FUNC_ARG if ctx or data is NULL
- PUF_READ_E if sz < WC_PUF_RAW_BYTES

Example

```
byte testSram[WC_PUF_RAW_BYTES];
wc_PufSetTestData(&ctx, testSram, sizeof(testSram));
```

B.48 Algorithms - RIPEMD

B.47.2.13 function wc_PufSetTestData

B.48.1 Functions

	Name
int	**wc_InitRipeMd**(RipeMd * ripemd)This function initializes a ripemd structure by initializing ripemd's digest, buffer, loLen and hiLen.
int	**wc_RipeMdUpdate**(RipeMd * ripemd, const byte * data, word32 len)This function generates the RipeMd digest of the data input and stores the result in the ripemd->digest buffer. After running wc_RipeMdUpdate, one should compare the generated ripemd->digest to a known authentication tag to verify the authenticity of a message.
int	**wc_RipeMdFinal**(RipeMd * ripemd, byte * hash)This function copies the computed digest into hash. If there is a partial unhashed block, this method will pad the block with 0s, and include that block's round in the digest before copying to hash. State of ripemd is reset.

B.48.2 Functions Documentation

```
int wc_InitRipeMd(
    RipeMd * ripemd
)
```

This function initializes a ripemd structure by initializing ripemd's digest, buffer, loLen and hiLen.

Parameters:

- **ripemd** pointer to the ripemd structure to initialize

See:

- wc_RipeMdUpdate
- wc_RipeMdFinal

Return:

- 0 returned on successful execution of the function. The RipeMd structure is initialized.
- BAD_FUNC_ARG returned if the RipeMd structure is NULL.

Example

```
RipeMd md;
int ret;
ret = wc_InitRipeMd(&md);
if (ret != 0) {
    // Failure case.
}
```

```
int wc_RipeMdUpdate(
    RipeMd * ripemd,
    const byte * data,
    word32 len
)
```

This function generates the RipeMd digest of the data input and stores the result in the ripemd->digest buffer. After running wc_RipeMdUpdate, one should compare the generated ripemd->digest to a known authentication tag to verify the authenticity of a message.

Parameters:

- **ripemd** pointer to the ripemd structure to be initialized with wc_InitRipeMd
- **data** data to be hashed
- **len** sizeof data in bytes

See:

- wc_InitRipeMd
- wc_RipeMdFinal

Return:

- 0 Returned on successful execution of the function.
- BAD_FUNC_ARG Returned if the RipeMd structure is NULL or if data is NULL and len is not zero. This function should execute if data is NULL and len is 0.

Example

```
const byte* data; // The data to be hashed
....
RipeMd md;
int ret;
ret = wc_InitRipeMd(&md);
if (ret == 0) {
ret = wc_RipeMdUpdate(&md, plain, sizeof(plain));
if (ret != 0) {
// Failure case …
```

```
int wc_RipeMdFinal(
    RipeMd * ripemd,
    byte * hash
)
```

This function copies the computed digest into hash. If there is a partial unhashed block, this method will pad the block with 0s, and include that block's round in the digest before copying to hash. State of ripemd is reset.

Parameters:

- **ripemd** pointer to the ripemd structure to be initialized with wc_InitRipeMd, and containing hashes from wc_RipeMdUpdate. State will be reset
- **hash** buffer to copy digest to. Should be RIPEMD_DIGEST_SIZE bytes

See: none

Return:

- 0 Returned on successful execution of the function. The state of the RipeMd structure has been reset.
- BAD_FUNC_ARG Returned if the RipeMd structure or hash parameters are NULL.

Example

```
RipeMd md;
int ret;
byte   digest[RIPEMD_DIGEST_SIZE];
const byte* data; // The data to be hashed
...
ret = wc_InitRipeMd(&md);
if (ret == 0) {
ret = wc_RipeMdUpdate(&md, plain, sizeof(plain));
    if (ret != 0) {
        // RipeMd Update Failure Case.
}
ret = wc_RipeMdFinal(&md, digest);
if (ret != 0) {
    // RipeMd Final Failure Case.
}...
```

B.49 Algorithms - RSA

B.48.2.3 function wc_RipeMdFinal

B.49.1 Functions

	Name
int	**wc_SetKeyUsage**(Cert * cert, const char * value)This function allows you to set the key usage using a comma delimited string of tokens. Accepted tokens are: digitalSignature, nonRepudiation, contentCommitment, keyCertSign, cRLSign, dataEncipherment, keyAgreement, keyEncipherment, encipherOnly, decipherOnly. Example: “digitalSignature,nonRepudiation” nonRepudiation and contentCommitment are for the same usage.
int	**wc_RsaPublicKeyDecode_ex**(const byte * input, word32 * inOutIdx, word32 inSz, const byte ** n, word32 * nSz, const byte ** e, word32 * eSz)Decodes RSA public key and extracts modulus and exponent.
int	**wc_RsaPublicKeyDerSize**(RsaKey * key, int with_header)Calculates DER encoded RSA public key size.
int	**wc_RsaPrivateKeyValidate**(const byte * input, word32 * inOutIdx, int * keySz, word32 inSz)Validates DER encoded RSA private key format. This function validates the ASN.1 syntax and structure of the RSA private key (sequences, integer tags, and lengths) without loading the key values into an RsaKey structure. It does not perform mathematical validation of the RSA key parameters (e.g., checking if p and q are prime, or if the key components satisfy RSA mathematical relationships).
int	**wc_InitRsaKey**(RsaKey * key, void * heap)This function initializes a provided RsaKey struct. It also takes in a heap identifier, for use with user defined memory overrides (see XMALLOC, XFREE, XREALLOC).
int	**wc_InitRsaKey_Id**(RsaKey * key, unsigned char * id, int len, void * heap, int devId)This function initializes a provided RsaKey struct. The id and len are used to identify the key on the device while the devId identifies the device. It also takes in a heap identifier, for use with user defined memory overrides (see XMALLOC, XFREE, XREALLOC).
int	**wc_RsaSetRNG**(RsaKey * key, WC_RNG * rng)This function associates RNG with Key. It is needed when WC_RSA_BLINDING is enabled.
int	**wc_FreeRsaKey**(RsaKey * key)This function frees a provided RsaKey struct using mp_clear.

	Name
int	**wc_RsaDirect**(const byte * in, word32 inLen, byte * out, word32 * outSz, RsaKey * key, int type, WC_RNG * rng)Function that does the RSA operation directly with no padding. The input size must match key size. Typically this is used when padding is already done on the RSA input.
int	**wc_RsaPublicEncrypt**(const byte * in, word32 inLen, byte * out, word32 outLen, RsaKey * key, WC_RNG * rng)This function encrypts a message from in and stores the result in out. It requires an initialized public key and a random number generator. As a side effect, this function will return the bytes written to out in outLen.
int	**wc_RsaPrivateDecryptInline**(byte * in, word32 inLen, byte ** out, RsaKey * key)This functions is utilized by the wc_RsaPrivateDecrypt function for decrypting.
int	**wc_RsaPrivateDecrypt**(const byte * in, word32 inLen, byte * out, word32 outLen, RsaKey * key)This functions provides private RSA decryption.
int	**wc_RsaSSL_Sign**(const byte * in, word32 inLen, byte * out, word32 outLen, RsaKey * key, WC_RNG * rng)Signs the provided array with the private key.
int	**wc_RsaSSL_VerifyInline**(byte * in, word32 inLen, byte ** out, RsaKey * key)Used to verify that the message was signed by RSA key. The output uses the same byte array as the input.
int	**wc_RsaSSL_Verify**(const byte * in, word32 inLen, byte * out, word32 outLen, RsaKey * key)Used to verify that the message was signed by key.
int	**wc_RsaPSS_Sign**(const byte * in, word32 inLen, byte * out, word32 outLen, enum wc_HashType hash, int mgf, RsaKey * key, WC_RNG * rng)Signs the provided array with the private key.
int	**wc_RsaPSS_Verify**(const byte * in, word32 inLen, byte * out, word32 outLen, enum wc_HashType hash, int mgf, RsaKey * key)Decrypt input signature to verify that the message was signed by key. The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.

	Name
int	**wc_RsaPSS_VerifyInline**(byte * in, word32 inLen, byte ** out, enum wc_HashType hash, int mgf, RsaKey * key)Decrypt input signature to verify that the message was signed by RSA key. The output uses the same byte array as the input. The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.
int	**wc_RsaPSS_VerifyCheck**(const byte * in, word32 inLen, byte * out, word32 outLen, const byte * digest, word32 digestLen, enum wc_HashType hash, int mgf, RsaKey * key)Verify the message signed with RSA-PSS. Salt length is equal to hash length. The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.
int	**wc_RsaPSS_VerifyCheck_ex**(byte * in, word32 inLen, byte * out, word32 outLen, const byte * digest, word32 digestLen, enum wc_HashType hash, int mgf, int saltLen, RsaKey * key)Verify the message signed with RSA-PSS. The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.
int	**wc_RsaPSS_VerifyCheckInline**(byte * in, word32 inLen, byte ** out, const byte * digest, word32 digentLen, enum wc_HashType hash, int mgf, RsaKey * key)Verify the message signed with RSA-PSS. The input buffer is reused for the output buffer. Salt length is equal to hash length.
int	**wc_RsaPSS_VerifyCheckInline_ex**(byte * in, word32 inLen, byte ** out, const byte * digest, word32 digentLen, enum wc_HashType hash, int mgf, int saltLen, RsaKey * key)Verify the message signed with RSA-PSS. The input buffer is reused for the output buffer. The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.
int	**wc_RsaPSS_CheckPadding**(const byte * in, word32 inLen, const byte * sig, word32 sigSz, enum wc_HashType hashType)Checks the PSS data to ensure that the signature matches. Salt length is equal to hash length. The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.
int	**wc_RsaPSS_CheckPadding_ex**(const byte * in, word32 inLen, const byte * sig, word32 sigSz, enum wc_HashType hashType, int saltLen, int bits)Checks the PSS data to ensure that the signature matches. Salt length is equal to hash length.

	Name
int	**wc_RsaEncryptSize**(const RsaKey * key)Returns the encryption size for the provided key structure.
int	**wc_RsaPrivateKeyDecode**(const byte * input, word32 * inOutIdx, RsaKey * key, word32 inSz)This function parses a DER-formatted RSA private key, extracts the private key and stores it in the given RsaKey structure. It also sets the distance parsed in idx.
int	**wc_RsaPublicKeyDecode**(const byte * input, word32 * inOutIdx, RsaKey * key, word32 inSz)This function parses a DER-formatted RSA public key, extracts the public key and stores it in the given RsaKey structure. It also sets the distance parsed in idx.
int	**wc_RsaPublicKeyDecodeRaw**(const byte * n, word32 nSz, const byte * e, word32 eSz, RsaKey * key)This function decodes the raw elements of an RSA public key, taking in the public modulus (n) and exponent (e). It stores these raw elements in the provided RsaKey structure, allowing one to use them in the encryption/decryption process.
int	**wc_RsaKeyToDer**(RsaKey * key, byte * output, word32 outLen)This function converts an RsaKey key to DER format. The result is written to output and it returns the number of bytes written.
int	**wc_RsaPublicEncrypt_ex**(const byte * in, word32 inLen, byte * out, word32 outLen, RsaKey * key, WC_RNG * rng, int type, enum wc_HashType hash, int mgf, byte * label, word32 labelSz)This function performs RSA encrypt while allowing the choice of which padding to use.
int	**wc_RsaPrivateDecrypt_ex**(const byte * in, word32 inLen, byte * out, word32 outLen, RsaKey * key, int type, enum wc_HashType hash, int mgf, byte * label, word32 labelSz)This function uses RSA to decrypt a message and gives the option of what padding type.
int	**wc_RsaPrivateDecryptInline_ex**(byte * in, word32 inLen, byte ** out, RsaKey * key, int type, enum wc_HashType hash, int mgf, byte * label, word32 labelSz)This function uses RSA to decrypt a message inline and gives the option of what padding type. The in buffer will contain the decrypted message after being called and the out byte pointer will point to the location in the “in” buffer where the plain text is.

	Name
int	**wc_RsaFlattenPublicKey**(const RsaKey * key, byte * e, word32 * eSz, byte * n, word32 * nSz)Flattens the RsaKey structure into individual elements (e, n) used for the RSA algorithm.
int	**wc_RsaKeyToPublicDer**(RsaKey * key, byte * output, word32 outLen)Convert Rsa Public key to DER format. Writes to output, and returns count of bytes written.
int	**wc_RsaKeyToPublicDer_ex**(RsaKey * key, byte * output, word32 outLen, int with_header)Convert RSA Public key to DER format. Writes to output, and returns count of bytes written. If with_header is 0 then only the (seq + n + e) is returned in ASN.1 DER format and will exclude the header.
int	**wc_MakeRsaKey**(RsaKey * key, int size, long e, WC_RNG * rng)This function generates a RSA private key of length size (in bits) and given exponent (e). It then stores this key in the provided RsaKey structure, so that it may be used for encryption/decryption. A secure number to use for e is 65537. size is required to be greater than or equal to RSA_MIN_SIZE and less than or equal to RSA_MAX_SIZE. For this function to be available, the option WOLFSSL_KEY_GEN must be enabled at compile time. This can be accomplished with –enable-keygen if using ./configure.
int	**wc_RsaSetNonBlock**(RsaKey * key, RsaNb * nb)This function sets the non-blocking RSA context. When a RsaNb context is set it enables fast math based non-blocking exptmod, which splits the RSA function into many smaller operations. Enabled when WC_RSA_NONBLOCK is defined.
int	**wc_RsaSetNonBlockTime**(RsaKey * key, word32 maxBlockUs, word32 cpuMHz)This function configures the maximum amount of blocking time in microseconds. It uses a pre_computed table (see tfm.c exptModNbInst) along with the CPU speed in megahertz to determine if the next operation can be completed within the maximum blocking time provided. Enabled when WC_RSA_NONBLOCK_TIME is defined.
int	**wc_InitRsaKey_ex**(RsaKey * key, void * heap, int devId)Initializes RSA key with heap and device ID.

	Name
RsaKey *	**wc_NewRsaKey**(void * heap, int devId, int * result_code)Allocates and initializes new RSA key. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.
int	**wc_DeleteRsaKey**(RsaKey * key, RsaKey ** key_p)Deletes and frees RSA key. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.
int	**wc_InitRsaKey_Label**(RsaKey * key, const char * label, void * heap, int devId)Initializes RSA key with label.
int	**wc_CheckRsaKey**(RsaKey * key)Checks RSA key validity.
int	**wc_RsaUseKeyId**(RsaKey * key, word32 keyId, word32 flags)Uses key ID for hardware RSA.
int	**wc_RsaGetKeyId**(RsaKey * key, word32 * keyId)Gets key ID from hardware RSA key.
int	**wc_RsaFunction**(const byte * in, word32 inLen, byte * out, word32 * outLen, int type, RsaKey * key, WC_RNG * rng)Performs RSA operation.
int	**wc_RsaPSS_Sign_ex**(const byte * in, word32 inLen, byte * out, word32 outLen, enum wc_HashType hash, int mgf, int saltLen, RsaKey * key, WC_RNG * rng)Signs with RSA-PSS extended options.
int	**wc_RsaSSL_Verify_ex**(const byte * in, word32 inLen, byte * out, word32 outLen, RsaKey * key, int pad_type)Verifies RSA signature with padding type.
int	**wc_RsaSSL_Verify_ex2**(const byte * in, word32 inLen, byte * out, word32 outLen, RsaKey * key, int pad_type, enum wc_HashType hash)Verifies RSA signature with hash type.
int	**wc_RsaPSS_VerifyInline_ex**(byte * in, word32 inLen, byte ** out, enum wc_HashType hash, int mgf, int saltLen, RsaKey * key)Verifies RSA-PSS inline with extended options.
int	**wc_RsaPSS_Verify_ex**(const byte * in, word32 inLen, byte * out, word32 outLen, enum wc_HashType hash, int mgf, int saltLen, RsaKey * key)Verifies RSA-PSS with extended options.
int	**wc_RsaPSS_CheckPadding_ex2**(const byte * in, word32 inLen, const byte * sig, word32 sigSz, enum wc_HashType hashType, int saltLen, int bits, void * heap)Checks RSA-PSS padding with extended options.

	Name
int	**wc_RsaExportKey**(const RsaKey * key, byte * e, word32 * eSz, byte * n, word32 * nSz, byte * d, word32 * dSz, byte * p, word32 * pSz, byte * q, word32 * qSz)Exports RSA key components.
int	**wc_CheckProbablePrime_ex**(const byte * p, word32 pSz, const byte * q, word32 qSz, const byte * e, word32 eSz, int nlen, int * isPrime, WC_RNG * rng)Checks probable prime with extended options.
int	**wc_CheckProbablePrime**(const byte * p, word32 pSz, const byte * q, word32 qSz, const byte * e, word32 eSz, int nlen, int * isPrime)Checks probable prime.
int	**wc_RsaPad_ex**(const byte * input, word32 inputLen, byte * pkcsBlock, word32 pkcsBlockLen, byte padValue, WC_RNG * rng, int padType, enum wc_HashType hType, int mgf, byte * optLabel, word32 labelLen, int saltLen, int bits, void * heap)Pads data with extended options.
int	**wc_RsaUnPad_ex**(byte * pkcsBlock, word32 pkcsBlockLen, byte ** out, byte padValue, int padType, enum wc_HashType hType, int mgf, byte * optLabel, word32 labelLen, int saltLen, int bits, void * heap)Unpads data with extended options.
int	**wc_RsaPrivateKeyDecodeRaw**(const byte * n, word32 nSz, const byte * e, word32 eSz, const byte * d, word32 dSz, const byte * u, word32 uSz, const byte * p, word32 pSz, const byte * q, word32 qSz, const byte * dP, word32 dPSz, const byte * dQ, word32 dQSz, RsaKey * key)Decodes raw RSA private key.

B.49.2 Functions Documentation

```
int wc_SetKeyUsage(
    Cert * cert,
    const char * value
)
```

This function allows you to set the key usage using a comma delimited string of tokens. Accepted tokens are: digitalSignature, nonRepudiation, contentCommitment, keyCertSign, cRLSign, dataEncipherment, keyAgreement, keyEncipherment, encipherOnly, decipherOnly. Example: "digitalSignature,nonRepudiation" nonRepudiation and contentCommitment are for the same usage.

Parameters:

- **cert** Pointer to initialized Cert structure.
- **value** Comma delimited string of tokens to set usage.

See:

- wc_InitCert
- wc_MakeRsaKey

Return:

- 0 Success
- BAD_FUNC_ARG Returned when either arg is null.
- MEMORY_E Returned when there is an error allocating memory.
- KEYUSAGE_E Returned if an unrecognized token is entered.

Example

```
Cert cert;
wc_InitCert(&cert);

if(wc_SetKeyUsage(&cert, "cRLSign,keyCertSign") != 0)
{
    // Handle error
}
```

```
int wc_RsaPublicKeyDecode_ex(
    const byte * input,
    word32 * inOutIdx,
    word32 inSz,
    const byte ** n,
    word32 * nSz,
    const byte ** e,
    word32 * eSz
)
```

Decodes RSA public key and extracts modulus and exponent.

Parameters:

- **input** DER encoded RSA public key buffer
- **inOutIdx** Pointer to index in buffer
- **inSz** Size of input buffer
- **n** Pointer to modulus pointer
- **nSz** Pointer to modulus size
- **e** Pointer to exponent pointer
- **eSz** Pointer to exponent size

See: wc_RsaPublicKeyDecode

Return:

- 0 on success
- negative on error

Example

```
const byte* n;
const byte* e;
word32 nSz, eSz, idx = 0;
int ret = wc_RsaPublicKeyDecode_ex(derBuf, &idx, derSz,
                                   &n, &nSz, &e, &eSz);
```

```
int wc_RsaPublicKeyDerSize(
    RsaKey * key,
    int with_header
)
```

Calculates DER encoded RSA public key size.

Parameters:

- **key** RSA key structure
- **with_header** Include sequence header if non-zero

See: wc_RsaKeyToDer

Return:

- Size on success
- negative on error

Example

```
RsaKey key;
int derSz = wc_RsaPublicKeyDerSize(&key, 1);
```

```
int wc_RsaPrivateKeyValidate(
    const byte * input,
    word32 * inOutIdx,
    int * keySz,
    word32 inSz
)
```

Validates DER encoded RSA private key format. This function validates the ASN.1 syntax and structure of the RSA private key (sequences, integer tags, and lengths) without loading the key values into an RsaKey structure. It does not perform mathematical validation of the RSA key parameters (e.g., checking if p and q are prime, or if the key components satisfy RSA mathematical relationships).

Parameters:

- **input** DER encoded RSA private key buffer
- **inOutIdx** Pointer to index in buffer (updated on success)

- **keySz** Pointer to store modulus size in bytes
- **inSz** Size of input buffer

See: wc_RsaPrivateKeyDecode

Return:

- 0 on success (valid ASN.1 structure)
- ASN_PARSE_E if ASN.1 parsing fails
- ASN_RSA_KEY_E if RSA key structure is invalid
- BAD_FUNC_ARG if parameters are invalid

Example

```
word32 idx = 0;
int keySz;
int ret = wc_RsaPrivateKeyValidate(derBuf, &idx, &keySz,
                                   derSz);
if (ret == 0) {
    // ASN.1 structure is valid, keySz contains modulus size
}
```

```
int wc_InitRsaKey(
    RsaKey * key,
    void * heap
)
```

This function initializes a provided RsaKey struct. It also takes in a heap identifier, for use with user defined memory overrides (see XMALLOC, XFREE, XREALLOC).

Parameters:

- **key** pointer to the RsaKey structure to initialize
- **heap** pointer to a heap identifier, for use with memory overrides, allowing custom handling of memory allocation. This heap will be the default used when allocating memory for use with this RSA object

See:

- wc_FreeRsaKey
- wc_RsaSetRNG

Return:

- 0 Returned upon successfully initializing the RSA structure for use with encryption and decryption
- BAD_FUNC_ARGS Returned if the RSA key pointer evaluates to NULL

The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.

Example

```
RsaKey enc;
int ret;
ret = wc_InitRsaKey(&enc, NULL); // not using heap hint. No custom memory
if ( ret != 0 ) {
    // error initializing RSA key
}
```

```
int wc_InitRsaKey_Id(
    RsaKey * key,
    unsigned char * id,
    int len,
    void * heap,
    int devId
)
```

This function initializes a provided RsaKey struct. The id and len are used to identify the key on the device while the devId identifies the device. It also takes in a heap identifier, for use with user defined memory overrides (see XMALLOC, XFREE, XREALLOC).

Parameters:

- **key** pointer to the RsaKey structure to initialize
- **id** identifier of key on device
- **len** length of identifier in bytes
- **heap** pointer to a heap identifier, for use with memory overrides, allowing custom handling of memory allocation. This heap will be the default used when allocating memory for use with this RSA object
- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID (-2) if not used

See:

- wc_InitRsaKey
- wc_FreeRsaKey
- wc_RsaSetRNG

Return:

- 0 Returned upon successfully initializing the RSA structure for use with encryption and decryption
- BAD_FUNC_ARGS Returned if the RSA key pointer evaluates to NULL, or if id is NULL while len is positive
- BUFFER_E Returned if len is less than 0 or greater than RSA_MAX_ID_LEN.

Note: This API is only available when WOLF_PRIVATE_KEY_ID is defined, which is set for PKCS11 support.

The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.

Example

```
RsaKey enc;
unsigned char* id = (unsigned char*)"RSA2048";
```

```
int len = 7;
int devId = 1;
int ret;
ret = wc_CryptoDev_RegisterDevice(devId, wc_Pkcs11_CryptoDevCb,
                                  &token);
if ( ret != 0) {
    // error associating callback and token with device id
}
ret = wc_InitRsaKey_Id(&enc, id, len, NULL, devId); // not using heap hint
if ( ret != 0 ) {
    // error initializing RSA key
}
```

```
int wc_RsaSetRNG(
    RsaKey * key,
    WC_RNG * rng
)
```

This function associates RNG with Key. It is needed when WC_RSA_BLINDING is enabled.

Parameters:

- **key** pointer to the RsaKey structure to be associated
- **rng** pointer to the WC_RNG structure to associate with

See:

- wc_InitRsaKey
- wc_RsaSetRNG

Return:

- 0 Returned upon success
- BAD_FUNC_ARGS Returned if the RSA key, rng pointer evaluates to NULL

Example

```
ret = wc_InitRsaKey(&key, NULL);
if (ret == 0) {
    ret = wc_InitRng(&rng);
} else return -1;
if (ret == 0) {
    ret = wc_RsaSetRNG(&key, &rng);
```

```
int wc_FreeRsaKey(
    RsaKey * key
)
```

This function frees a provided RsaKey struct using mp_clear.

Parameters:

- **key** pointer to the RsaKey structure to free

See: wc_InitRsaKey

Return: 0 Returned upon successfully freeing the key

Example

```
RsaKey enc;
wc_InitRsaKey(&enc, NULL); // not using heap hint. No custom memory
... set key, do encryption

wc_FreeRsaKey(&enc);
```

```
int wc_RsaDirect(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outSz,
    RsaKey * key,
    int type,
    WC_RNG * rng
)
```

Function that does the RSA operation directly with no padding. The input size must match key size. Typically this is used when padding is already done on the RSA input.

Parameters:

- **in** buffer to do operation on
- **inLen** length of input buffer
- **out** buffer to hold results
- **outSz** gets set to size of result buffer. Should be passed in as length of out buffer. If the pointer "out" is null then outSz gets set to the expected buffer size needed and LENGTH_ONLY_E gets returned.
- **key** initialized RSA key to use for encrypt/decrypt
- **type** if using private or public key (RSA_PUBLIC_ENCRYPT, RSA_PUBLIC_DECRYPT, RSA_PRIVATE_ENCRYPT, RSA_PRIVATE_DECRYPT)
- **rng** initialized WC_RNG struct

See:

- wc_RsaPublicEncrypt
- wc_RsaPrivateDecrypt

Return:

- size On successfully encryption the size of the encrypted buffer is returned
- RSA_BUFFER_E RSA buffer error, output too small or input too large

Example

```
int ret;
WC_RNG rng;
RsaKey key;
byte  in[256];
byte out[256];
word32 outSz = (word32)sizeof(out);
…

ret = wc_RsaDirect(in, (word32)sizeof(in), out, &outSz, &key,
    RSA_PRIVATE_ENCRYPT, &rng);
if (ret < 0) {
    //handle error
}
```

```
int wc_RsaPublicEncrypt(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    RsaKey * key,
    WC_RNG * rng
)
```

This function encrypts a message from in and stores the result in out. It requires an initialized public key and a random number generator. As a side effect, this function will return the bytes written to out in outLen.

Parameters:

- **in** pointer to a buffer containing the input message to encrypt
- **inLen** the length of the message to encrypt
- **out** pointer to the buffer in which to store the output ciphertext
- **outLen** the length of the output buffer
- **key** pointer to the RsaKey structure containing the public key to use for encryption
- **rng** The RNG structure with which to generate random block padding

See: wc_RsaPrivateDecrypt

Return:

- Success Upon successfully encrypting the input message, returns the number of bytes written on success and less than zero for failure.
- BAD_FUNC_ARG Returned if any of the input parameters are invalid
- RSA_BUFFER_E Returned if the output buffer is too small to store the ciphertext
- RNG_FAILURE_E Returned if there is an error generating a random block using the provided RNG structure

- MP_INIT_E May be returned if there is an error in the math library used while encrypting the message
- MP_READ_E May be returned if there is an error in the math library used while encrypting the message
- MP_CMP_E May be returned if there is an error in the math library used while encrypting the message
- MP_INVMOD_E May be returned if there is an error in the math library used while encrypting the message
- MP_EXPTMOD_E May be returned if there is an error in the math library used while encrypting the message
- MP_MOD_E May be returned if there is an error in the math library used while encrypting the message
- MP_MUL_E May be returned if there is an error in the math library used while encrypting the message
- MP_ADD_E May be returned if there is an error in the math library used while encrypting the message
- MP_MULMOD_E May be returned if there is an error in the math library used while encrypting the message
- MP_TO_E May be returned if there is an error in the math library used while encrypting the message
- MP_MEM May be returned if there is an error in the math library used while encrypting the message
- MP_ZERO_E May be returned if there is an error in the math library used while encrypting the message

Example

```
RsaKey pub;
int ret = 0;
byte n[] = { // initialize with received n component of public key };
byte e[] = { // initialize with received e component of public key };
byte msg[] = { // initialize with plaintext of message to encrypt };
byte cipher[256]; // 256 bytes is large enough to store 2048 bit RSA
ciphertext

wc_InitRsaKey(&pub, NULL); // not using heap hint. No custom memory
wc_RsaPublicKeyDecodeRaw(n, sizeof(n), e, sizeof(e), &pub);
// initialize with received public key parameters
ret = wc_RsaPublicEncrypt(msg, sizeof(msg), out, sizeof(out), &pub, &rng);
if ( ret != 0 ) {
    // error encrypting message
}
```

```
int wc_RsaPrivateDecryptInline(
    byte * in,
    word32 inLen,
    byte ** out,
    RsaKey * key
)
```

This functions is utilized by the wc_RsaPrivateDecrypt function for decrypting.

Parameters:

- **in** The byte array to be decrypted.
- **inLen** The length of in.
- **out** The byte array for the decrypted data to be stored.
- **key** The key to use for decryption.

See: wc_RsaPrivateDecrypt

Return:

- Success Length of decrypted data.
- RSA_PAD_E RsaUnPad error, bad formatting

Example

none

```
int wc_RsaPrivateDecrypt(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    RsaKey * key
)
```

This functions provides private RSA decryption.

Parameters:

- **in** The byte array to be decrypted.
- **inLen** The length of in.
- **out** The byte array for the decrypted data to be stored.
- **outLen** The length of out.
- **key** The key to use for decryption.

See:

- RsaUnPad
- wc_RsaFunction
- wc_RsaPrivateDecryptInline

Return:

- Success length of decrypted data.
- MEMORY_E -125, out of memory error
- BAD_FUNC_ARG -173, Bad function argument provided

Example

```
ret = wc_RsaPublicEncrypt(in, inLen, out, sizeof(out), &key, &rng);
if (ret < 0) {
    return -1;
}
ret = wc_RsaPrivateDecrypt(out, ret, plain, sizeof(plain), &key);
if (ret < 0) {
    return -1;
}
```

```
int wc_RsaSSL_Sign(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    RsaKey * key,
    WC_RNG * rng
)
```

Signs the provided array with the private key.

Parameters:

- **in** The byte array to be encrypted.
- **inLen** The length of in.
- **out** The byte array for the encrypted data to be stored.
- **outLen** The length of out.
- **key** The key to use for encryption.
- **RNG** The RNG struct to use for random number purposes.

See: wc_RsaPad

Return: RSA_BUFFER_E: -131, RSA buffer error, output too small or input too large

Example

```
ret = wc_RsaSSL_Sign(in, inLen, out, sizeof(out), &key, &rng);
if (ret < 0) {
    return -1;
}
memset(plain, 0, sizeof(plain));
ret = wc_RsaSSL_Verify(out, ret, plain, sizeof(plain), &key);
if (ret < 0) {
    return -1;
}
if (ret != inLen) {
    return -1;
}
if (XMEMCMP(in, plain, ret) != 0) {
```

```
    return -1;
}



int wc_RsaSSL_VerifyInline(
    byte * in,
    word32 inLen,
    byte ** out,
    RsaKey * key
)
```

Used to verify that the message was signed by RSA key. The output uses the same byte array as the input.

Parameters:

- **in** Byte array to be decrypted.
- **inLen** Length of the buffer input.
- **out** Pointer to a pointer for decrypted information.
- **key** RsaKey to use.

See:

- wc_RsaSSL_Verify
- wc_RsaSSL_Sign

Return:

- 0 Length of the digest.
- <0 An error occurred.

Example

```
RsaKey key;
WC_RNG rng;
int ret = 0;
long e = 65537; // standard value to use for exponent
wc_InitRsaKey(&key, NULL); // not using heap hint. No custom memory
wc_InitRng(&rng);
wc_MakeRsaKey(&key, 2048, e, &rng);

byte in[] = { // Initialize with some RSA encrypted information }
byte* out;
if(wc_RsaSSL_VerifyInline(in, sizeof(in), &out, &key) < 0)
{
    // handle error
}
```

```
int wc_RsaSSL_Verify(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    RsaKey * key
)
```

Used to verify that the message was signed by key.

Parameters:

- **in** The byte array to be decrypted.
- **inLen** The length of in.
- **out** The byte array for the decrypted data to be stored.
- **outLen** The length of out.
- **key** The key to use for verification.

See: wc_RsaSSL_Sign

Return:

- Success Length of digest on no error.
- MEMORY_E memory exception.

Example

```
ret = wc_RsaSSL_Sign(in, inLen, out, sizeof(out), &key, &rng);
if (ret < 0) {
    return -1;
}
memset(plain, 0, sizeof(plain));
ret = wc_RsaSSL_Verify(out, ret, plain, sizeof(plain), &key);
if (ret < 0) {
    return -1;
}
if (ret != inLen) {
    return -1;
}
if (XMEMCMP(in, plain, ret) != 0) {
    return -1;
}
```

```
int wc_RsaPSS_Sign(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    enum wc_HashType hash,
    int mgf,
    RsaKey * key,
```

```
    WC_RNG * rng
)
```

Signs the provided array with the private key.

Parameters:

- **in** The byte array to be encrypted.
- **inLen** The length of in.
- **out** The byte array for the encrypted data to be stored.
- **outLen** The length of out.
- **hash** The hash type to be in message
- **mgf** Mask Generation Function Identifiers
- **key** The key to use for verification.

See:

- wc_RsaPSS_Verify
- wc_RsaSetRNG

Return: RSA_BUFFER_E: -131, RSA buffer error, output too small or input too large

Example

```
ret = wc_InitRsaKey(&key, NULL);
if (ret == 0) {
    ret = wc_InitRng(&rng);
} else return -1;
if (ret == 0) {
    ret = wc_RsaSetRNG(&key, &rng);
} else return -1;
if (ret == 0) {
        ret = wc_MakeRsaKey(&key, 2048, WC_RSA_EXPONENT, &rng);
} else return -1;

ret = wc_RsaPSS_Sign((byte*)szMessage, (word32)XSTRLEN(szMessage)+1,
        pSignature, sizeof(pSignature),
        WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key, &rng);
if (ret > 0 ){
    sz = ret;
} else return -1;

ret = wc_RsaPSS_Verify(pSignature, sz, pt, outLen,
        WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key);
if (ret <= 0)return -1;

wc_FreeRsaKey(&key);
wc_FreeRng(&rng);
```

```
int wc_RsaPSS_Verify(
    const byte * in,
```

```
    word32 inLen,
    byte * out,
    word32 outLen,
    enum wc_HashType hash,
    int mgf,
    RsaKey * key
)
```

Decrypt input signature to verify that the message was signed by key. The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.

Parameters:

- **in** The byte array to be decrypted.
- **inLen** The length of in.
- **out** The byte array for the decrypted data to be stored.
- **outLen** The length of out.
- **hash** The hash type to be in message
- **mgf** Mask Generation Function Identifiers
- **key** The key to use for verification.

See:

- wc_RsaPSS_Sign
- wc_RsaPSS_VerifyInline
- wc_RsaPSS_CheckPadding
- wc_RsaSetRNG

Return:

- Success Length of text on no error.
- MEMORY_E memory exception.
- MP_EXPTMOD_E - When using fastmath and FP_MAX_BITS not set to at least 2 times the keySize (Example when using 4096-bit key set FP_MAX_BITS to 8192 or greater value)

Example

```
ret = wc_InitRsaKey(&key, NULL);
if (ret == 0) {
    ret = wc_InitRng(&rng);
} else return -1;
if (ret == 0) {
    ret = wc_RsaSetRNG(&key, &rng);
} else return -1;
if (ret == 0) {
        ret = wc_MakeRsaKey(&key, 2048, WC_RSA_EXPONENT, &rng);
} else return -1;
ret = wc_RsaPSS_Sign((byte*)szMessage, (word32)XSTRLEN(szMessage)+1,
        pSignature, sizeof(pSignature),
        WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key, &rng);
if (ret > 0 ){
    sz = ret;
```

```
} else return -1;

ret = wc_RsaPSS_Verify(pSignature, sz, pt, outLen,
        WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key);
if (ret <= 0)return -1;

wc_FreeRsaKey(&key);
wc_FreeRng(&rng);



int wc_RsaPSS_VerifyInline(
    byte * in,
    word32 inLen,
    byte ** out,
    enum wc_HashType hash,
    int mgf,
    RsaKey * key
)
```

Decrypt input signature to verify that the message was signed by RSA key. The output uses the same byte array as the input. The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.

Parameters:

- **in** Byte array to be decrypted.
- **inLen** Length of the buffer input.
- **out** Pointer to address containing the PSS data.
- **hash** The hash type to be in message
- **mgf** Mask Generation Function Identifiers
- **key** RsaKey to use.

See:

- wc_RsaPSS_Verify
- wc_RsaPSS_Sign
- wc_RsaPSS_VerifyCheck
- wc_RsaPSS_VerifyCheck_ex
- wc_RsaPSS_VerifyCheckInline
- wc_RsaPSS_VerifyCheckInline_ex
- wc_RsaPSS_CheckPadding
- wc_RsaPSS_CheckPadding_ex
- wc_RsaSetRNG

Return:

- 0 Length of text.
- <0 An error occurred.

Example

```
ret = wc_InitRsaKey(&key, NULL);
if (ret == 0) {
    ret = wc_InitRng(&rng);
} else return -1;
if (ret == 0) {
    ret = wc_RsaSetRNG(&key, &rng);
} else return -1;
if (ret == 0) {
        ret = wc_MakeRsaKey(&key, 2048, WC_RSA_EXPONENT, &rng);
} else return -1;
ret = wc_RsaPSS_Sign(digest, digestSz, pSignature, pSignatureSz,
        WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key, &rng);
if (ret > 0 ){
    sz = ret;
} else return -1;

ret = wc_RsaPSS_VerifyInline(pSignature, sz, pt,
        WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key);
if (ret <= 0)return -1;

wc_FreeRsaKey(&key);
wc_FreeRng(&rng);
```

```
int wc_RsaPSS_VerifyCheck(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    const byte * digest,
    word32 digestLen,
    enum wc_HashType hash,
    int mgf,
    RsaKey * key
)
```

Verify the message signed with RSA-PSS. Salt length is equal to hash length. The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.

Parameters:

- **in** The byte array to be decrypted.
- **inLen** The length of in.
- **out** Pointer to address containing the PSS data. Zeroed when a crypto callback device reported only a verdict (see
- **outLen** The length of out.
- **digest** Hash of the data that is being verified.
- **digestLen** Length of hash.
- **hash** Hash algorithm.
- **mgf** Mask generation function.

- **key** Public RSA key.

See:

- wc_RsaPSS_Sign
- wc_RsaPSS_Verify
- wc_RsaPSS_VerifyCheck_ex
- wc_RsaPSS_VerifyCheckInline
- wc_RsaPSS_VerifyCheckInline_ex
- wc_RsaPSS_CheckPadding
- wc_RsaPSS_CheckPadding_ex
- wc_RsaSetRNG

Return:

- the length of the PSS data on success and negative indicates failure. A crypto callback device that returns recovered data fills out and the return is that length. A device that reports only a verdict recovers nothing: out is zeroed and the return is the length the data would have been, so callers must not read out on that path.
- MEMORY_E memory exception.
-).

Example

```
ret = wc_InitRsaKey(&key, NULL);
if (ret == 0) {
    ret = wc_InitRng(&rng);
} else return -1;
if (ret == 0) {
    ret = wc_RsaSetRNG(&key, &rng);
} else return -1;
if (ret == 0) {
        ret = wc_MakeRsaKey(&key, 2048, WC_RSA_EXPONENT, &rng);
} else return -1;

if (ret == 0) {
    digestSz = wc_HashGetDigestSize(WC_HASH_TYPE_SHA256);
    ret = wc_Hash(WC_HASH_TYPE_SHA256, message, sz, digest, digestSz);
} else return -1;

if (ret == 0) {
    ret = wc_RsaPSS_Sign(digest, digestSz, pSignature, pSignatureSz,
            WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key, &rng);
    if (ret > 0 ){
        sz = ret;
    } else return -1;
} else return -1;
if (ret == 0) {
    ret = wc_RsaPSS_VerifyCheck(pSignature, sz, pt, outLen,
            digest, digestSz, WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key);
        if (ret <= 0) return -1;
} else return -1;
```

```
wc_FreeRsaKey(&key);
wc_FreeRng(&rng);
```

```
int wc_RsaPSS_VerifyCheck_ex(
    byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    const byte * digest,
    word32 digestLen,
    enum wc_HashType hash,
    int mgf,
    int saltLen,
    RsaKey * key
)
```

Verify the message signed with RSA-PSS. The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.

Parameters:

- **in** The byte array to be decrypted.
- **inLen** The length of in.
- **out** Pointer to address containing the PSS data.
- **outLen** The length of out.
- **digest** Hash of the data that is being verified.
- **digestLen** Length of hash.
- **hash** Hash algorithm.
- **mgf** Mask generation function.
- **saltLen** Length of salt used. RSA_PSS_SALT_LEN_DEFAULT (-1) indicates salt length is the same as the hash length. RSA_PSS_SALT_LEN_DISCOVER indicates salt length is determined from the data.
- **key** Public RSA key.

See:

- wc_RsaPSS_Sign
- wc_RsaPSS_Verify
- wc_RsaPSS_VerifyCheck
- wc_RsaPSS_VerifyCheckInline
- wc_RsaPSS_VerifyCheckInline_ex
- wc_RsaPSS_CheckPadding
- wc_RsaPSS_CheckPadding_ex
- wc_RsaSetRNG

Return:

- the length of the PSS data on success and negative indicates failure.
- MEMORY_E memory exception.

Example

```
ret = wc_InitRsaKey(&key, NULL);
if (ret == 0) {
    ret = wc_InitRng(&rng);
} else return -1;
if (ret == 0) {
    ret = wc_RsaSetRNG(&key, &rng);
} else return -1;
if (ret == 0) {
        ret = wc_MakeRsaKey(&key, 2048, WC_RSA_EXPONENT, &rng);
} else return -1;

if (ret == 0) {
    digestSz = wc_HashGetDigestSize(WC_HASH_TYPE_SHA256);
    ret = wc_Hash(WC_HASH_TYPE_SHA256, message, sz, digest, digestSz);
} else return -1;

if (ret == 0) {
    ret = wc_RsaPSS_Sign(digest, digestSz, pSignature, pSignatureSz,
            WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key, &rng);
    if (ret > 0 ){
        sz = ret;
    } else return -1;
} else return -1;
if (ret == 0) {
    ret = wc_RsaPSS_VerifyCheck_ex(pSignature, sz, pt, outLen,
            digest, digestSz, WC_HASH_TYPE_SHA256, WC_MGF1SHA256, saltLen,
 ↪  &key);
        if (ret <= 0) return -1;
} else return -1;

wc_FreeRsaKey(&key);
wc_FreeRng(&rng);
```

```
int wc_RsaPSS_VerifyCheckInline(
    byte * in,
    word32 inLen,
    byte ** out,
    const byte * digest,
    word32 digentLen,
    enum wc_HashType hash,
    int mgf,
    RsaKey * key
)
```

Verify the message signed with RSA-PSS. The input buffer is reused for the output buffer. Salt length is equal to hash length.

Parameters:

- **in** The byte array to be decrypted.

- **inLen** The length of in.
- **out** The byte array for the decrypted data to be stored. Set to NULL when a crypto callback device reported only a verdict (see
- **digest** Hash of the data that is being verified.
- **digestLen** Length of hash.
- **hash** The hash type to be in message
- **mgf** Mask Generation Function Identifiers
- **key** The key to use for verification.

See:

- wc_RsaPSS_Sign
- wc_RsaPSS_Verify
- wc_RsaPSS_VerifyCheck
- wc_RsaPSS_VerifyCheck_ex
- wc_RsaPSS_VerifyCheckInline_ex
- wc_RsaPSS_CheckPadding
- wc_RsaPSS_CheckPadding_ex
- wc_RsaSetRNG

Return:

- the length of the PSS data on success and negative indicates failure. A crypto callback device that returns recovered data points *out into in and the return is that length. A device that reports only a verdict recovers nothing:* out is set to NULL though the return stays positive, so callers must check *out before dereferencing it.
-).

The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.

Example

```
ret = wc_InitRsaKey(&key, NULL);
if (ret == 0) {
    ret = wc_InitRng(&rng);
} else return -1;
if (ret == 0) {
    ret = wc_RsaSetRNG(&key, &rng);
} else return -1;
if (ret == 0) {
        ret = wc_MakeRsaKey(&key, 2048, WC_RSA_EXPONENT, &rng);
} else return -1;

if (ret == 0) {
    digestSz = wc_HashGetDigestSize(WC_HASH_TYPE_SHA256);
    ret = wc_Hash(WC_HASH_TYPE_SHA256, message, sz, digest, digestSz);
} else return -1;

if (ret == 0) {
    ret = wc_RsaPSS_Sign(digest, digestSz, pSignature, pSignatureSz,
            WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key, &rng);
    if (ret > 0 ){
        sz = ret;
```

```
    } else return -1;
} else return -1;
if (ret == 0) {
    ret = wc_RsaPSS_VerifyCheckInline(pSignature, sz, pt,
            digest, digestSz, WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key);
        if (ret <= 0) return -1;
} else return -1;

wc_FreeRsaKey(&key);
wc_FreeRng(&rng);
```

```
int wc_RsaPSS_VerifyCheckInline_ex(
    byte * in,
    word32 inLen,
    byte ** out,
    const byte * digest,
    word32 digentLen,
    enum wc_HashType hash,
    int mgf,
    int saltLen,
    RsaKey * key
)
```

Verify the message signed with RSA-PSS. The input buffer is reused for the output buffer. The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.

Parameters:

- **in** The byte array to be decrypted.
- **inLen** The length of in.
- **out** The byte array for the decrypted data to be stored.
- **digest** Hash of the data that is being verified.
- **digestLen** Length of hash.
- **hash** The hash type to be in message
- **mgf** Mask Generation Function Identifiers
- **saltLen** Length of salt used. RSA_PSS_SALT_LEN_DEFAULT (-1) indicates salt length is the same as the hash length. RSA_PSS_SALT_LEN_DISCOVER indicates salt length is determined from the data.
- **key** The key to use for verification.

See:

- wc_RsaPSS_Sign
- wc_RsaPSS_Verify
- wc_RsaPSS_VerifyCheck
- wc_RsaPSS_VerifyCheck_ex
- wc_RsaPSS_VerifyCheckInline
- wc_RsaPSS_CheckPadding
- wc_RsaPSS_CheckPadding_ex
- wc_RsaSetRNG

Return: the length of the PSS data on success and negative indicates failure.

Example

```
ret = wc_InitRsaKey(&key, NULL);
if (ret == 0) {
    ret = wc_InitRng(&rng);
} else return -1;
if (ret == 0) {
    ret = wc_RsaSetRNG(&key, &rng);
} else return -1;
if (ret == 0) {
        ret = wc_MakeRsaKey(&key, 2048, WC_RSA_EXPONENT, &rng);
} else return -1;

if (ret == 0) {
    digestSz = wc_HashGetDigestSize(WC_HASH_TYPE_SHA256);
    ret = wc_Hash(WC_HASH_TYPE_SHA256, message, sz, digest, digestSz);
} else return -1;

if (ret == 0) {
    ret = wc_RsaPSS_Sign(digest, digestSz, pSignature, pSignatureSz,
            WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key, &rng);
    if (ret > 0 ){
        sz = ret;
    } else return -1;
} else return -1;
if (ret == 0) {
    ret = wc_RsaPSS_VerifyCheckInline_ex(pSignature, sz, pt,
            digest, digestSz, WC_HASH_TYPE_SHA256, WC_MGF1SHA256, saltLen,
 ↪  &key);
        if (ret <= 0) return -1;
} else return -1;

wc_FreeRsaKey(&key);
wc_FreeRng(&rng);
```

```
int wc_RsaPSS_CheckPadding(
    const byte * in,
    word32 inLen,
    const byte * sig,
    word32 sigSz,
    enum wc_HashType hashType
)
```

Checks the PSS data to ensure that the signature matches. Salt length is equal to hash length. The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.

Parameters:

- **in** Hash of the data that is being verified.
- **inSz** Length of hash.
- **sig** Buffer holding PSS data.

- **sigSz** Size of PSS data.
- **hashType** Hash algorithm.

See:

- wc_RsaPSS_Sign
- wc_RsaPSS_Verify
- wc_RsaPSS_VerifyInline
- wc_RsaPSS_VerifyCheck
- wc_RsaPSS_VerifyCheck_ex
- wc_RsaPSS_VerifyCheckInline
- wc_RsaPSS_VerifyCheckInline_ex
- wc_RsaPSS_CheckPadding_ex
- wc_RsaSetRNG

Return:

- BAD_PADDING_E when the PSS data is invalid, BAD_FUNC_ARG when NULL is passed in to in or sig or inSz is not the same as the hash algorithm length and 0 on success.
- MEMORY_E memory exception.

Example

```
ret = wc_InitRsaKey(&key, NULL);
if (ret == 0) {
    ret = wc_InitRng(&rng);
} else return -1;
if (ret == 0) {
    ret = wc_RsaSetRNG(&key, &rng);
} else return -1;
if (ret == 0) {
        ret = wc_MakeRsaKey(&key, 2048, WC_RSA_EXPONENT, &rng);
} else return -1;
if (ret == 0) {
    digestSz = wc_HashGetDigestSize(WC_HASH_TYPE_SHA256);
    ret = wc_Hash(WC_HASH_TYPE_SHA256, message, sz, digest, digestSz);
} else return -1;
ret = wc_RsaPSS_Sign(digest, digestSz, pSignature, sizeof(pSignature),
        WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key, &rng);
if (ret > 0 ){
    sz = ret;
} else return -1;

verify = wc_RsaPSS_Verify(pSignature, sz, out, outLen,
        WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key);
if (verify <= 0)return -1;

ret = wc_RsaPSS_CheckPadding(digest, digestSz, out, verify, hash);

wc_FreeRsaKey(&key);
wc_FreeRng(&rng);
```

```
int wc_RsaPSS_CheckPadding_ex(
    const byte * in,
    word32 inLen,
    const byte * sig,
    word32 sigSz,
    enum wc_HashType hashType,
    int saltLen,
    int bits
)
```

Checks the PSS data to ensure that the signature matches. Salt length is equal to hash length.

Parameters:

- **in** Hash of the data that is being verified.
- **inSz** Length of hash.
- **sig** Buffer holding PSS data.
- **sigSz** Size of PSS data.
- **hashType** Hash algorithm.
- **saltLen** Length of salt used. RSA_PSS_SALT_LEN_DEFAULT (-1) indicates salt length is the same as the hash length. RSA_PSS_SALT_LEN_DISCOVER indicates salt length is determined from the data.
- **bits** Can be used to calculate salt size in FIPS case

See:

- wc_RsaPSS_Sign
- wc_RsaPSS_Verify
- wc_RsaPSS_VerifyInline
- wc_RsaPSS_VerifyCheck
- wc_RsaPSS_VerifyCheck_ex
- wc_RsaPSS_VerifyCheckInline
- wc_RsaPSS_VerifyCheckInline_ex
- wc_RsaPSS_CheckPadding

Return:

- BAD_PADDING_E when the PSS data is invalid, BAD_FUNC_ARG when NULL is passed in to in or sig or inSz is not the same as the hash algorithm length and 0 on success.
- MEMORY_E memory exception.

Example

```
ret = wc_InitRsaKey(&key, NULL);
if (ret == 0) {
    ret = wc_InitRng(&rng);
} else return -1;
if (ret == 0) {
    ret = wc_RsaSetRNG(&key, &rng);
} else return -1;
if (ret == 0) {
        ret = wc_MakeRsaKey(&key, 2048, WC_RSA_EXPONENT, &rng);
```

```
} else return -1;
if (ret == 0) {
    digestSz = wc_HashGetDigestSize(WC_HASH_TYPE_SHA256);
    ret = wc_Hash(WC_HASH_TYPE_SHA256, message, sz, digest, digestSz);
} else return -1;
ret = wc_RsaPSS_Sign(digest, digestSz, pSignature, sizeof(pSignature),
        WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key, &rng);
if (ret > 0 ){
    sz = ret;
} else return -1;

verify = wc_RsaPSS_Verify(pSignature, sz, out, outLen,
        WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key);
if (verify <= 0)return -1;

ret = wc_RsaPSS_CheckPadding_ex(digest, digestSz, out, verify, hash, saltLen,
 ↪  0);

wc_FreeRsaKey(&key);
wc_FreeRng(&rng);
```

```
int wc_RsaEncryptSize(
    const RsaKey * key
)
```

Returns the encryption size for the provided key structure.

Parameters:

- **key** The key to use for verification.

See:

- wc_InitRsaKey
- wc_InitRsaKey_ex
- wc_MakeRsaKey

Return: Success Encryption size for the provided key structure.

Example

```
int sz = wc_RsaEncryptSize(&key);
```

```
int wc_RsaPrivateKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    RsaKey * key,
    word32 inSz
)
```

This function parses a DER-formatted RSA private key, extracts the private key and stores it in the given RsaKey structure. It also sets the distance parsed in idx.

Parameters:

- **input** pointer to the buffer containing the DER formatted private key to decode
- **inOutIdx** pointer to the index in the buffer at which the key begins (usually 0). As a side effect of this function, inOutIdx will store the distance parsed through the input buffer
- **key** pointer to the RsaKey structure in which to store the decoded private key
- **inSz** size of the input buffer

See:

- wc_RsaPublicKeyDecode
- wc_MakeRsaKey

Return:

- 0 Returned upon successfully parsing the private key from the DER encoded input
- ASN_PARSE_E Returned if there is an error parsing the private key from the input buffer. This may happen if the input private key is not properly formatted according to ASN.1 standards
- ASN_RSA_KEY_E Returned if there is an error reading the private key elements of the RSA key input

Example

```
RsaKey enc;
word32 idx = 0;
int ret = 0;
byte der[] = { // initialize with DER-encoded RSA private key };

wc_InitRsaKey(&enc, NULL); // not using heap hint. No custom memory
ret = wc_RsaPrivateKeyDecode(der, &idx, &enc, sizeof(der));
if( ret != 0 ) {
    // error parsing private key
}
```

```
int wc_RsaPublicKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    RsaKey * key,
    word32 inSz
)
```

This function parses a DER-formatted RSA public key, extracts the public key and stores it in the given RsaKey structure. It also sets the distance parsed in idx.

Parameters:

- **input** pointer to the buffer containing the input DER-encoded RSA public key to decode

- **inOutIdx** pointer to the index in the buffer at which the key begins (usually 0). As a side effect of this function, inOutIdx will store the distance parsed through the input buffer
- **key** pointer to the RsaKey structure in which to store the decoded public key
- **inSz** size of the input buffer

See: wc_RsaPublicKeyDecodeRaw

Return:

- 0 Returned upon successfully parsing the public key from the DER encoded input
- ASN_PARSE_E Returned if there is an error parsing the public key from the input buffer. This may happen if the input public key is not properly formatted according to ASN.1 standards
- ASN_OBJECT_ID_E Returned if the ASN.1 Object ID does not match that of a RSA public key
- ASN_EXPECT_0_E Returned if the input key is not correctly formatted according to ASN.1 standards
- ASN_BITSTR_E Returned if the input key is not correctly formatted according to ASN.1 standards
- ASN_RSA_KEY_E Returned if there is an error reading the public key elements of the RSA key input

Example

```
RsaKey pub;
word32 idx = 0;
int ret = 0;
byte der[] = { // initialize with DER-encoded RSA public key };

wc_InitRsaKey(&pub, NULL); // not using heap hint. No custom memory
ret = wc_RsaPublicKeyDecode(der, &idx, &pub, sizeof(der));
if( ret != 0 ) {
    // error parsing public key
}
```

```
int wc_RsaPublicKeyDecodeRaw(
    const byte * n,
    word32 nSz,
    const byte * e,
    word32 eSz,
    RsaKey * key
)
```

This function decodes the raw elements of an RSA public key, taking in the public modulus (n) and exponent (e). It stores these raw elements in the provided RsaKey structure, allowing one to use them in the encryption/decryption process.

Parameters:

- **n** pointer to a buffer containing the raw modulus parameter of the public RSA key
- **nSz** size of the buffer containing n
- **e** pointer to a buffer containing the raw exponent parameter of the public RSA key
- **eSz** size of the buffer containing e
- **key** pointer to the RsaKey struct to initialize with the provided public key elements

See: wc_RsaPublicKeyDecode

Return:

- 0 Returned upon successfully decoding the raw elements of the public key into the RsaKey structure
- BAD_FUNC_ARG Returned if any of the input arguments evaluates to NULL
- MP_INIT_E Returned if there is an error initializing an integer for use with the multiple precision integer (mp_int) library
- ASN_GETINT_E Returned if there is an error reading one of the provided RSA key elements, n or e

Example

```
RsaKey pub;
int ret = 0;
byte n[] = { // initialize with received n component of public key };
byte e[] = { // initialize with received e component of public key };

wc_InitRsaKey(&pub, NULL); // not using heap hint. No custom memory
ret = wc_RsaPublicKeyDecodeRaw(n, sizeof(n), e, sizeof(e), &pub);
if( ret != 0 ) {
    // error parsing public key elements
}
```

```
int wc_RsaKeyToDer(
    RsaKey * key,
    byte * output,
    word32 outLen
)
```

This function converts an RsaKey key to DER format. The result is written to output and it returns the number of bytes written.

Parameters:

- **key** Initialized RsaKey structure.
- **output** Pointer to output buffer.
- **outLen** Size of output buffer.

See:

- wc_RsaKeyToPublicDer
- wc_InitRsaKey
- wc_MakeRsaKey
- wc_InitRng

Return:

-

0 Success, number of bytes written.

- BAD_FUNC_ARG Returned if key or output is null, or if key->type is not RSA_PRIVATE, or if outLen isn't large enough for output buffer.
- MEMORY_E Returned if there is an error allocating memory.

Example

```
byte* der;
// Allocate memory for der
int derSz = // Amount of memory allocated for der;
RsaKey key;
WC_RNG rng;
long e = 65537; // standard value to use for exponent
ret = wc_MakeRsaKey(&key, 2048, e, &rng); // generate 2048 bit long
private key
wc_InitRsaKey(&key, NULL);
wc_InitRng(&rng);
if(wc_RsaKeyToDer(&key, der, derSz) != 0)
{
    // Handle the error thrown
}
```

```
int wc_RsaPublicEncrypt_ex(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    RsaKey * key,
    WC_RNG * rng,
    int type,
    enum wc_HashType hash,
    int mgf,
    byte * label,
    word32 labelSz
)
```

This function performs RSA encrypt while allowing the choice of which padding to use.

Parameters:

- **in** pointer to the buffer for encryption
- **inLen** length of the buffer to encrypt
- **out** encrypted msg created
- **outLen** length of buffer available to hold encrypted msg
- **key** initialized RSA key struct
- **rng** initialized WC_RNG struct
- **type** type of padding to use (WC_RSA_OAEP_PAD or WC_RSA_PKCSV15_PAD)
- **hash** type of hash to use (choices can be found in hash.h)
- **mgf** type of mask generation function to use
- **label** an optional label to associate with encrypted message

- **labelSz** size of the optional label used

See:

- wc_RsaPublicEncrypt
- wc_RsaPrivateDecrypt_ex

Return:

- size On successfully encryption the size of the encrypted buffer is returned
- RSA_BUFFER_E RSA buffer error, output too small or input too large

Example

```
WC_RNG rng;
RsaKey key;
byte in[] = “I use Turing Machines to ask questions”
byte out[256];
int ret;
…

ret = wc_RsaPublicEncrypt_ex(in, sizeof(in), out, sizeof(out), &key, &rng,
WC_RSA_OAEP_PAD, WC_HASH_TYPE_SHA, WC_MGF1SHA1, NULL, 0);
if (ret < 0) {
    //handle error
}
```

```
int wc_RsaPrivateDecrypt_ex(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    RsaKey * key,
    int type,
    enum wc_HashType hash,
    int mgf,
    byte * label,
    word32 labelSz
)
```

This function uses RSA to decrypt a message and gives the option of what padding type.

Parameters:

- **in** pointer to the buffer for decryption
- **inLen** length of the buffer to decrypt
- **out** decrypted msg created
- **outLen** length of buffer available to hold decrypted msg
- **key** initialized RSA key struct
- **type** type of padding to use (WC_RSA_OAEP_PAD or WC_RSA_PKCSV15_PAD)

- **hash** type of hash to use (choices can be found in hash.h)
- **mgf** type of mask generation function to use
- **label** an optional label to associate with encrypted message
- **labelSz** size of the optional label used

See: none

Return:

- size On successful decryption, the size of the decrypted message is returned.
- MEMORY_E Returned if not enough memory on system to malloc a needed array.
- BAD_FUNC_ARG Returned if a bad argument was passed into the function.

Example

```
WC_RNG rng;
RsaKey key;
byte in[] = “I use Turing Machines to ask questions”
byte out[256];
byte plain[256];
int ret;
…
ret = wc_RsaPublicEncrypt_ex(in, sizeof(in), out, sizeof(out), &key,
&rng, WC_RSA_OAEP_PAD, WC_HASH_TYPE_SHA, WC_MGF1SHA1, NULL, 0);
if (ret < 0) {
    //handle error
}
…
ret = wc_RsaPrivateDecrypt_ex(out, ret, plain, sizeof(plain), &key,
WC_RSA_OAEP_PAD, WC_HASH_TYPE_SHA, WC_MGF1SHA1, NULL, 0);

if (ret < 0) {
    //handle error
}
```

```
int wc_RsaPrivateDecryptInline_ex(
    byte * in,
    word32 inLen,
    byte ** out,
    RsaKey * key,
    int type,
    enum wc_HashType hash,
    int mgf,
    byte * label,
    word32 labelSz
)
```

This function uses RSA to decrypt a message inline and gives the option of what padding type. The in buffer will contain the decrypted message after being called and the out byte pointer will point to the location in the “in” buffer where the plain text is.

Parameters:

- **in** pointer to the buffer for decryption
- **inLen** length of the buffer to decrypt
- **out** pointer to location of decrypted message in “in” buffer
- **key** initialized RSA key struct
- **type** type of padding to use (WC_RSA_OAEP_PAD or WC_RSA_PKCSV15_PAD)
- **hash** type of hash to use (choices can be found in hash.h)
- **mgf** type of mask generation function to use
- **label** an optional label to associate with encrypted message
- **labelSz** size of the optional label used

See: none

Return:

- size On successful decryption, the size of the decrypted message is returned.
- MEMORY_E: Returned if not enough memory on system to malloc a needed array.
- RSA_PAD_E: Returned if an error in the padding was encountered.
- BAD_PADDING_E: Returned if an error happened during parsing past padding.
- BAD_FUNC_ARG: Returned if a bad argument was passed into the function.

Example

```
WC_RNG rng;
RsaKey key;
byte in[] = “I use Turing Machines to ask questions”
byte out[256];
byte* plain;
int ret;
…
ret = wc_RsaPublicEncrypt_ex(in, sizeof(in), out, sizeof(out), &key,
&rng, WC_RSA_OAEP_PAD, WC_HASH_TYPE_SHA, WC_MGF1SHA1, NULL, 0);

if (ret < 0) {
    //handle error
}
…
ret = wc_RsaPrivateDecryptInline_ex(out, ret, &plain, &key,
WC_RSA_OAEP_PAD, WC_HASH_TYPE_SHA, WC_MGF1SHA1, NULL, 0);

if (ret < 0) {
    //handle error
}
```

```
int wc_RsaFlattenPublicKey(
    const RsaKey * key,
    byte * e,
    word32 * eSz,
    byte * n,
    word32 * nSz
)
```

Flattens the RsaKey structure into individual elements (e, n) used for the RSA algorithm.

Parameters:

- **key** The key to use for verification.
- **e** a buffer for the value of e. e is a large positive integer in the RSA modular arithmetic operation.
- **eSz** the size of the e buffer.
- **n** a buffer for the value of n. n is a large positive integer in the RSA modular arithmetic operation.
- **nSz** the size of the n buffer.

See:

- wc_InitRsaKey
- wc_InitRsaKey_ex
- wc_MakeRsaKey

Return:

- 0 Returned if the function executed normally, without error.
- BAD_FUNC_ARG: Returned if any of the parameters are passed in with a null value.
- RSA_BUFFER_E: Returned if the e or n buffers passed in are not the correct size.
- MP_MEM: Returned if an internal function has memory errors.
- MP_VAL: Returned if an internal function argument is not valid.

Example

```
Rsa key; // A valid RSA key.
byte e[ buffer sz E.g. 256 ];
byte n[256];
int ret;
word32 eSz = sizeof(e);
word32 nSz = sizeof(n);
...
ret = wc_RsaFlattenPublicKey(&key, e, &eSz, n, &nSz);
if (ret != 0) {
    // Failure case.
}
```

```
int wc_RsaKeyToPublicDer(
    RsaKey * key,
    byte * output,
    word32 outLen
)
```

Convert Rsa Public key to DER format. Writes to output, and returns count of bytes written.

Parameters:

- **key** The RSA key structure to convert.
- **output** Output buffer to hold DER. (if NULL will return length only)

- **outLen** Length of buffer.

See:

- wc_RsaPublicKeyDerSize
- wc_RsaKeyToPublicDer_ex
- wc_InitRsaKey

Return:

- 0 Success, number of bytes written.
- BAD_FUNC_ARG Returned if key or output is null.
- MEMORY_E Returned when an error allocating memory occurs.
- <0 Error

Example

```
RsaKey key;

wc_InitRsaKey(&key, NULL);
// Use key

const int BUFFER_SIZE = 1024; // Some adequate size for the buffer
byte output[BUFFER_SIZE];
if (wc_RsaKeyToPublicDer(&key, output, sizeof(output)) != 0) {
    // Handle Error
}
```

```
int wc_RsaKeyToPublicDer_ex(
    RsaKey * key,
    byte * output,
    word32 outLen,
    int with_header
)
```

Convert RSA Public key to DER format. Writes to output, and returns count of bytes written. If with_header is 0 then only the (seq + n + e) is returned in ASN.1 DER format and will exclude the header.

Parameters:

- **key** The RSA key structure to convert.
- **output** Output buffer to hold DER. (if NULL will return length only)
- **outLen** Length of buffer.

See:

- wc_RsaPublicKeyDerSize
- wc_RsaKeyToPublicDer
- wc_InitRsaKey

Return:

- 0 Success, number of bytes written.
- BAD_FUNC_ARG Returned if key or output is null.
- MEMORY_E Returned when an error allocating memory occurs.
- <0 Error

Example

```
RsaKey key;

wc_InitRsaKey(&key, NULL);
// Use key

const int BUFFER_SIZE = 1024; // Some adequate size for the buffer
byte output[BUFFER_SIZE];
if (wc_RsaKeyToPublicDer_ex(&key, output, sizeof(output), 0) != 0) {
    // Handle Error
}
```

```
int wc_MakeRsaKey(
    RsaKey * key,
    int size,
    long e,
    WC_RNG * rng
)
```

This function generates a RSA private key of length size (in bits) and given exponent (e). It then stores this key in the provided RsaKey structure, so that it may be used for encryption/decryption. A secure number to use for e is 65537. size is required to be greater than or equal to RSA_MIN_SIZE and less than or equal to RSA_MAX_SIZE. For this function to be available, the option WOLFSSL_KEY_GEN must be enabled at compile time. This can be accomplished with –enable-keygen if using ./configure.

Parameters:

- **key** pointer to the RsaKey structure in which to store the generated private key
- **size** desired key length, in bits. Required to be greater than RSA_MIN_SIZE and less than RSA_MAX_SIZE
- **e** exponent parameter to use for generating the key. A secure choice is 65537
- **rng** pointer to an RNG structure to use for random number generation while making the key

See: none

Return:

- 0 Returned upon successfully generating a RSA private key
- BAD_FUNC_ARG Returned if any of the input arguments are NULL, the size parameter falls outside of the necessary bounds, or e is incorrectly chosen
- RNG_FAILURE_E Returned if there is an error generating a random block using the provided RNG structure
- MP_INIT_E
- MP_READ_E May be May be returned if there is an error in the math library used while generating the RSA key returned if there is an error in the math library used while generating the RSA key
- MP_CMP_E May be returned if there is an error in the math library used while generating the RSA key
- MP_INVMOD_E May be returned if there is an error in the math library used while generating the RSA key
- MP_EXPTMOD_E May be returned if there is an error in the math library used while generating the RSA key
- MP_MOD_E May be returned if there is an error in the math library used while generating the RSA key
- MP_MUL_E May be returned if there is an error in the math library used while generating the RSA key
- MP_ADD_E May be returned if there is an error in the math library used while generating the RSA key
- MP_MULMOD_E May be returned if there is an error in the math library used while generating the RSA key
- MP_TO_E May be returned if there is an error in the math library used while generating the RSA key
- MP_MEM May be returned if there is an error in the math library used while generating the RSA key
- MP_ZERO_E May be returned if there is an error in the math library used while generating the RSA key

Example

```
RsaKey priv;
WC_RNG rng;
int ret = 0;
long e = 65537; // standard value to use for exponent

wc_InitRsaKey(&priv, NULL); // not using heap hint. No custom memory
wc_InitRng(&rng);
// generate 2048 bit long private key
ret = wc_MakeRsaKey(&priv, 2048, e, &rng);
if( ret != 0 ) {
    // error generating private key
}
```

```
int wc_RsaSetNonBlock(
    RsaKey * key,
    RsaNb * nb
)
```

This function sets the non-blocking RSA context. When a RsaNb context is set it enables fast math

based non-blocking exptmod, which splits the RSA function into many smaller operations. Enabled when WC_RSA_NONBLOCK is defined.

Parameters:

- **key** The RSA key structure
- **nb** The RSA non-blocking structure for this RSA key to use.

See: wc_RsaSetNonBlockTime

Return:

- 0 Success
- BAD_FUNC_ARG Returned if key or nb is null.

Example

```
int ret, count = 0;
RsaKey key;
RsaNb  nb;

wc_InitRsaKey(&key, NULL);

// Enable non-blocking RSA mode - provide context
ret = wc_RsaSetNonBlock(key, &nb);
if (ret != 0)
    return ret;

do {
    ret = wc_RsaSSL_Sign(in, inLen, out, outSz, key, rng);
    count++; // track number of would blocks
    if (ret == FP_WOULDBLOCK) {
        // do "other" work here
    }
} while (ret == FP_WOULDBLOCK);
if (ret < 0) {
    return ret;
}

printf("RSA non-block sign: size %d, %d times\n", ret, count);
```

```
int wc_RsaSetNonBlockTime(
    RsaKey * key,
    word32 maxBlockUs,
    word32 cpuMHz
)
```

This function configures the maximum amount of blocking time in microseconds. It uses a pre-computed table (see tfm.c exptModNbInst) along with the CPU speed in megahertz to determine if the next operation can be completed within the maximum blocking time provided. Enabled when WC_RSA_NONBLOCK_TIME is defined.

Parameters:

- **key** The RSA key structure.
- **maxBlockUs** Maximum time to block microseconds.
- **cpuMHz** CPU speed in megahertz.

See: wc_RsaSetNonBlock

Return:

- 0 Success
- BAD_FUNC_ARG Returned if key is null or wc_RsaSetNonBlock was not previously called and key->nb is null.

Example

```
RsaKey key;
RsaNb  nb;

wc_InitRsaKey(&key, NULL);
wc_RsaSetNonBlock(key, &nb);
wc_RsaSetNonBlockTime(&key, 4000, 160); // Block Max = 4 ms, CPU = 160MHz
```

```
int wc_InitRsaKey_ex(
    RsaKey * key,
    void * heap,
    int devId
)
```

Initializes RSA key with heap and device ID.

Parameters:

- **key** RSA key structure
- **heap** Heap hint
- **devId** Device ID

See: wc_InitRsaKey

Return:

- 0 on success
- negative on error

Example

```
RsaKey key;
int ret = wc_InitRsaKey_ex(&key, NULL, INVALID_DEVID);
```

```
RsaKey * wc_NewRsaKey(
    void * heap,
    int devId,
    int * result_code
)
```

Allocates and initializes new RSA key. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.

Parameters:

- **heap** Heap hint
- **devId** Device ID
- **result_code** Result code pointer

See: wc_DeleteRsaKey

Return:

- RsaKey pointer on success
- NULL on failure

Note: This API is only available when WC_NO_CONSTRUCTORS is not defined. WC_NO_CONSTRUCTORS is automatically defined when WOLFSSL_NO_MALLOC is defined.

Example

```
int result;
RsaKey* key = wc_NewRsaKey(NULL, INVALID_DEVID, &result);
```

```
int wc_DeleteRsaKey(
    RsaKey * key,
    RsaKey ** key_p
)
```

Deletes and frees RSA key. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.

Parameters:

- **key** RSA key to delete
- **key_p** Pointer to key pointer

See: wc_NewRsaKey

Return:

- 0 on success
- negative on error

Note: This API is only available when WC_NO_CONSTRUCTORS is not defined. WC_NO_CONSTRUCTORS is automatically defined when WOLFSSL_NO_MALLOC is defined.

Example

```
RsaKey* key;
int ret = wc_DeleteRsaKey(key, &key);
```

```
int wc_InitRsaKey_Label(
    RsaKey * key,
    const char * label,
    void * heap,
    int devId
)
```

Initializes RSA key with label.

Parameters:

- **key** RSA key structure
- **label** Label string
- **heap** Heap hint
- **devId** Device ID

See: wc_InitRsaKey_ex

Return:

- 0 on success
- negative on error

Note: This API is only available when WOLF_PRIVATE_KEY_ID is defined, which is set for PKCS11 support.

Example

```
RsaKey key;
int ret = wc_InitRsaKey_Label(&key, "mykey", NULL,
                              INVALID_DEVID);
```

```
int wc_CheckRsaKey(
    RsaKey * key
)
```

Checks RSA key validity.

Parameters:

- **key** RSA key to check

See: wc_MakeRsaKey

Return:

- 0 on success
- negative on error

Example

```
RsaKey key;
int ret = wc_CheckRsaKey(&key);
```

```
int wc_RsaUseKeyId(
    RsaKey * key,
    word32 keyId,
    word32 flags
)
```

Uses key ID for hardware RSA.

Parameters:

- **key** RSA key
- **keyId** Key identifier
- **flags** Flags

See: wc_RsaGetKeyId

Return:

- 0 on success
- negative on error

Example

```
RsaKey key;
int ret = wc_RsaUseKeyId(&key, 1, 0);
```

```
int wc_RsaGetKeyId(
    RsaKey * key,
    word32 * keyId
)
```

Gets key ID from hardware RSA key.

Parameters:

- **key** RSA key

- **keyId** Key identifier pointer

See: wc_RsaUseKeyId

Return:

- 0 on success
- negative on error

Example

```
RsaKey key;
word32 keyId;
int ret = wc_RsaGetKeyId(&key, &keyId);
```

```
int wc_RsaFunction(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen,
    int type,
    RsaKey * key,
    WC_RNG * rng
)
```

Performs RSA operation.

Parameters:

- **in** Input buffer
- **inLen** Input length
- **out** Output buffer
- **outLen** Output length pointer
- **type** Operation type
- **key** RSA key
- **rng** Random number generator

See: wc_RsaPublicEncrypt

Return:

- 0 on success
- negative on error

Example

```
RsaKey key;
WC_RNG rng;
byte in[256], out[256];
word32 outLen = sizeof(out);
```

```
int ret = wc_RsaFunction(in, 256, out, &outLen,
                         RSA_PUBLIC_ENCRYPT, &key, &rng);
```

```
int wc_RsaPSS_Sign_ex(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    enum wc_HashType hash,
    int mgf,
    int saltLen,
    RsaKey * key,
    WC_RNG * rng
)
```

Signs with RSA-PSS extended options.

Parameters:

- **in** Input buffer
- **inLen** Input length
- **out** Output buffer
- **outLen** Output buffer size
- **hash** Hash type
- **mgf** MGF type
- **saltLen** Salt length
- **key** RSA key
- **rng** Random number generator

See: wc_RsaPSS_Sign

Return:

- Size of signature on success
- negative on error

Example

```
RsaKey key;
WC_RNG rng;
byte in[32], sig[256];
int ret = wc_RsaPSS_Sign_ex(in, 32, sig, sizeof(sig),
                            WC_HASH_TYPE_SHA256,
                            WC_MGF1SHA256, 32, &key, &rng);
```

```
int wc_RsaSSL_Verify_ex(
    const byte * in,
    word32 inLen,
    byte * out,
```

```
    word32 outLen,
    RsaKey * key,
    int pad_type
)
```

Verifies RSA signature with padding type.

Parameters:

- **in** Input signature
- **inLen** Signature length
- **out** Output buffer
- **outLen** Output buffer size
- **key** RSA key
- **pad_type** Padding type

See: wc_RsaSSL_Verify

Return:

- Size of decrypted data on success
- negative on error

Example

```
RsaKey key;
byte sig[256], out[256];
int ret = wc_RsaSSL_Verify_ex(sig, 256, out, sizeof(out),
                              &key, RSA_PKCS1_PADDING);
```

```
int wc_RsaSSL_Verify_ex2(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    RsaKey * key,
    int pad_type,
    enum wc_HashType hash
)
```

Verifies RSA signature with hash type.

Parameters:

- **in** Input signature
- **inLen** Signature length
- **out** Output buffer
- **outLen** Output buffer size
- **key** RSA key
- **pad_type** Padding type
- **hash** Hash type

See: wc_RsaSSL_Verify_ex

Return:

- Size of decrypted data on success
- negative on error

Example

```
RsaKey key;
byte sig[256], out[256];
int ret = wc_RsaSSL_Verify_ex2(sig, 256, out, sizeof(out),
                               &key, RSA_PKCS1_PADDING,
                               WC_HASH_TYPE_SHA256);
```

```
int wc_RsaPSS_VerifyInline_ex(
    byte * in,
    word32 inLen,
    byte ** out,
    enum wc_HashType hash,
    int mgf,
    int saltLen,
    RsaKey * key
)
```

Verifies RSA-PSS inline with extended options.

Parameters:

- **in** Input/output buffer
- **inLen** Input length
- **out** Output pointer
- **hash** Hash type
- **mgf** MGF type
- **saltLen** Salt length
- **key** RSA key

See: wc_RsaPSS_VerifyInline

Return:

- Size of verified data on success
- negative on error

Example

```
RsaKey key;
byte sig[256];
byte* out;
int ret = wc_RsaPSS_VerifyInline_ex(sig, 256, &out,
```

```
                                        WC_HASH_TYPE_SHA256,
                                        WC_MGF1SHA256, 32, &key);
```

```
int wc_RsaPSS_Verify_ex(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    enum wc_HashType hash,
    int mgf,
    int saltLen,
    RsaKey * key
)
```

Verifies RSA-PSS with extended options.

Parameters:

- **in** Input signature
- **inLen** Signature length
- **out** Output buffer
- **outLen** Output buffer size
- **hash** Hash type
- **mgf** MGF type
- **saltLen** Salt length
- **key** RSA key

See: wc_RsaPSS_Verify

Return:

- Size of verified data on success
- negative on error

Example

```
RsaKey key;
byte sig[256], out[256];
int ret = wc_RsaPSS_Verify_ex(sig, 256, out, sizeof(out),
                              WC_HASH_TYPE_SHA256,
                              WC_MGF1SHA256, 32, &key);
```

```
int wc_RsaPSS_CheckPadding_ex2(
    const byte * in,
    word32 inLen,
    const byte * sig,
    word32 sigSz,
    enum wc_HashType hashType,
    int saltLen,
```

```
    int bits,
    void * heap
)
```

Checks RSA-PSS padding with extended options.

Parameters:

- **in** Padded data
- **inLen** Padded data length
- **sig** Signature
- **sigSz** Signature size
- **hashType** Hash type
- **saltLen** Salt length
- **bits** Key size in bits
- **heap** Heap hint

See: wc_RsaPSS_CheckPadding_ex

Return:

- 0 on success
- negative on error

Example

```
byte padded[256], sig[256];
int ret = wc_RsaPSS_CheckPadding_ex2(padded, 256, sig, 256,
                                     WC_HASH_TYPE_SHA256, 32,
                                     2048, NULL);
```

```
int wc_RsaExportKey(
    const RsaKey * key,
    byte * e,
    word32 * eSz,
    byte * n,
    word32 * nSz,
    byte * d,
    word32 * dSz,
    byte * p,
    word32 * pSz,
    byte * q,
    word32 * qSz
)
```

Exports RSA key components.

Parameters:

- **key** RSA key
- **e** Public exponent buffer

- **eSz** Public exponent size pointer
- **n** Modulus buffer
- **nSz** Modulus size pointer
- **d** Private exponent buffer
- **dSz** Private exponent size pointer
- **p** Prime p buffer
- **pSz** Prime p size pointer
- **q** Prime q buffer
- **qSz** Prime q size pointer

See: wc_RsaFlattenPublicKey

Return:

- 0 on success
- negative on error

Example

```
RsaKey key;
byte e[3], n[256], d[256], p[128], q[128];
word32 eSz = 3, nSz = 256, dSz = 256, pSz = 128, qSz = 128;
int ret = wc_RsaExportKey(&key, e, &eSz, n, &nSz, d, &dSz,
                          p, &pSz, q, &qSz);
```

```
int wc_CheckProbablePrime_ex(
    const byte * p,
    word32 pSz,
    const byte * q,
    word32 qSz,
    const byte * e,
    word32 eSz,
    int nlen,
    int * isPrime,
    WC_RNG * rng
)
```

Checks probable prime with extended options.

Parameters:

- **p** Prime p buffer
- **pSz** Prime p size
- **q** Prime q buffer
- **qSz** Prime q size
- **e** Public exponent buffer
- **eSz** Public exponent size
- **nlen** Modulus length
- **isPrime** Prime result pointer
- **rng** Random number generator

See: wc_CheckProbablePrime

Return:

- 0 on success
- negative on error

Example

```
byte p[128], q[128], e[3];
int isPrime;
WC_RNG rng;
int ret = wc_CheckProbablePrime_ex(p, 128, q, 128, e, 3,
                                   2048, &isPrime, &rng);
```

```
int wc_CheckProbablePrime(
    const byte * p,
    word32 pSz,
    const byte * q,
    word32 qSz,
    const byte * e,
    word32 eSz,
    int nlen,
    int * isPrime
)
```

Checks probable prime.

Parameters:

- **p** Prime p buffer
- **pSz** Prime p size
- **q** Prime q buffer
- **qSz** Prime q size
- **e** Public exponent buffer
- **eSz** Public exponent size
- **nlen** Modulus length
- **isPrime** Prime result pointer

See: wc_CheckProbablePrime_ex

Return:

- 0 on success
- negative on error

Example

```
byte p[128], q[128], e[3];
int isPrime;
```

```
int ret = wc_CheckProbablePrime(p, 128, q, 128, e, 3, 2048,
                                &isPrime);
```

```
int wc_RsaPad_ex(
    const byte * input,
    word32 inputLen,
    byte * pkcsBlock,
    word32 pkcsBlockLen,
    byte padValue,
    WC_RNG * rng,
    int padType,
    enum wc_HashType hType,
    int mgf,
    byte * optLabel,
    word32 labelLen,
    int saltLen,
    int bits,
    void * heap
)
```

Pads data with extended options.

Parameters:

- **input** Input data
- **inputLen** Input length
- **pkcsBlock** Output padded block
- **pkcsBlockLen** Padded block size
- **padValue** Pad value
- **rng** Random number generator
- **padType** Padding type
- **hType** Hash type
- **mgf** MGF type
- **optLabel** Optional label
- **labelLen** Label length
- **saltLen** Salt length
- **bits** Key size in bits
- **heap** Heap hint

See: wc_RsaUnPad_ex

Return:

- 0 on success
- negative on error

Example

```
byte in[32], padded[256];
WC_RNG rng;
int ret = wc_RsaPad_ex(in, 32, padded, 256, 0x00, &rng,
```

```
                        RSA_BLOCK_TYPE_1,
                        WC_HASH_TYPE_SHA256, WC_MGF1SHA256,
                        NULL, 0, 32, 2048, NULL);
```

```
int wc_RsaUnPad_ex(
    byte * pkcsBlock,
    word32 pkcsBlockLen,
    byte ** out,
    byte padValue,
    int padType,
    enum wc_HashType hType,
    int mgf,
    byte * optLabel,
    word32 labelLen,
    int saltLen,
    int bits,
    void * heap
)
```

Unpads data with extended options.

Parameters:

- **pkcsBlock** Padded block
- **pkcsBlockLen** Padded block length
- **out** Output pointer
- **padValue** Pad value
- **padType** Padding type
- **hType** Hash type
- **mgf** MGF type
- **optLabel** Optional label
- **labelLen** Label length
- **saltLen** Salt length
- **bits** Key size in bits
- **heap** Heap hint

See: wc_RsaPad_ex

Return:

- Size of unpadded data on success
- negative on error

Example

```
byte padded[256];
byte* out;
int ret = wc_RsaUnPad_ex(padded, 256, &out, 0x00,
                         RSA_BLOCK_TYPE_1,
                         WC_HASH_TYPE_SHA256, WC_MGF1SHA256,
                         NULL, 0, 32, 2048, NULL);
```

```
int wc_RsaPrivateKeyDecodeRaw(
    const byte * n,
    word32 nSz,
    const byte * e,
    word32 eSz,
    const byte * d,
    word32 dSz,
    const byte * u,
    word32 uSz,
    const byte * p,
    word32 pSz,
    const byte * q,
    word32 qSz,
    const byte * dP,
    word32 dPSz,
    const byte * dQ,
    word32 dQSz,
    RsaKey * key
)
```

Decodes raw RSA private key.

Parameters:

- **n** Modulus buffer
- **nSz** Modulus size
- **e** Public exponent buffer
- **eSz** Public exponent size
- **d** Private exponent buffer
- **dSz** Private exponent size
- **u** Coefficient buffer
- **uSz** Coefficient size
- **p** Prime p buffer
- **pSz** Prime p size
- **q** Prime q buffer
- **qSz** Prime q size
- **dP** dP buffer
- **dPSz** dP size
- **dQ** dQ buffer
- **dQSz** dQ size
- **key** RSA key

See: wc_RsaPrivateKeyDecode

Return:

- 0 on success
- negative on error

Example

```
RsaKey key;
byte n[256], e[3], d[256], u[256], p[128], q[128];
byte dP[128], dQ[128];
int ret = wc_RsaPrivateKeyDecodeRaw(n, 256, e, 3, d, 256,
                                    u, 256, p, 128, q, 128,
                                    dP, 128, dQ, 128, &key);
```

B.50 Algorithms - SHA 128/224/256/384/512

B.49.2.58 function wc_RsaPrivateKeyDecodeRaw

B.50.1 Functions

	Name
int	**wc_ShaHash**(const byte * data, word32 len, byte * hash)Convenience function, handles all the hashing and places the result into hash.
int	**wc_Sha224Hash**(const byte * data, word32 len, byte * hash)Convenience function, handles all the hashing and places the result into hash.
int	**wc_Sha256Hash**(const byte * data, word32 len, byte * hash)Convenience function, handles all the hashing and places the result into hash.
int	**wc_Sha384Hash**(const byte * data, word32 len, byte * hash)Convenience function, handles all the hashing and places the result into hash.
int	**wc_Sha512Hash**(const byte * data, word32 len, byte * hash)Convenience function, handles all the hashing and places the result into hash.
int	**wc_Sha3_224Hash**(const byte * data, word32 len, byte * hash)Convenience function, handles all the hashing and places the result into hash.
int	**wc_Sha3_256Hash**(const byte * data, word32 len, byte * hash)Convenience function, handles all the hashing and places the result into hash.
int	**wc_Sha3_384Hash**(const byte * data, word32 len, byte * hash)Convenience function, handles all the hashing and places the result into hash.
int	**wc_Sha3_512Hash**(const byte * data, word32 len, byte * hash)Convenience function, handles all the hashing and places the result into hash.
int	**wc_Shake128Hash**(const byte * data, word32 len, byte * hash, word32 hashLen)Convenience function, handles all the hashing and places the result into hash.
int	**wc_Shake256Hash**(const byte * data, word32 len, byte * hash, word32 hashLen)Convenience function, handles all the hashing and places the result into hash.

	Name
int	**wc_InitSha**(wc_Sha * sha)This function initializes SHA. This is automatically called by wc_ShaHash.
int	**wc_ShaUpdate**(wc_Sha * sha, const byte * data, word32 len)Can be called to continually hash the provided byte array of length len.
int	**wc_ShaFinal**(wc_Sha * sha, byte * hash)Finalizes hashing of data. Result is placed into hash. Resets state of sha struct.
void	**wc_ShaFree**(wc_Sha * sha)Used to clean up memory used by an initialized Sha struct.
int	**wc_ShaGetHash**(wc_Sha * sha, byte * hash)Gets hash data. Result is placed into hash. Does not reset state of sha struct.
int	**wc_InitSha_ex**(wc_Sha * sha, void * heap, int devId)Initializes SHA with heap and device ID.
int	**wc_ShaFinalRaw**(wc_Sha * sha, byte * hash)Gets raw hash without finalizing.
int	**wc_ShaCopy**(wc_Sha * src, wc_Sha * dst)Copies SHA context.
int	**wc_ShaTransform**(wc_Sha * sha, const unsigned char * data)Transforms SHA block.
void	**wc_ShaSizeSet**(wc_Sha * sha, word32 len)Sets SHA size.
int	**wc_ShaSetFlags**(wc_Sha * sha, word32 flags)Sets SHA flags.
int	**wc_InitSha256**(wc_Sha256 * sha)This function initializes SHA256. This is automatically called by wc_Sha256Hash.
int	**wc_Sha256Update**(wc_Sha256 * sha, const byte * data, word32 len)Can be called to continually hash the provided byte array of length len.
int	**wc_Sha256Final**(wc_Sha256 * sha256, byte * hash)Finalizes hashing of data. Result is placed into hash. Resets state of sha256 struct.
void	**wc_Sha256Free**(wc_Sha256 * sha256)Resets the Sha256 structure. Note: this is only supported if you have WOLFSSL_TI_HASH defined.
int	**wc_Sha256Reset**(wc_Sha256 * sha256)Resets the sha256 structure to its freshly initialized state, continuing with its established allocations, heap hint, and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha256Free for teardown at end of life.

	Name
int	**wc_Sha256GetHash**(wc_Sha256 * sha256, byte * hash)Gets hash data. Result is placed into hash. Does not reset state of sha256 struct.
int	**wc_InitSha224**(wc_Sha224 * sha224)Used to initialize a Sha224 struct.
int	**wc_Sha224Update**(wc_Sha224 * sha224, const byte * data, word32 len)Can be called to continually hash the provided byte array of length len.
int	**wc_Sha224Final**(wc_Sha224 * sha224, byte * hash)Finalizes hashing of data. Result is placed into hash. Resets state of sha224 struct.
int	**wc_InitSha256_ex**(wc_Sha256 * sha, void * heap, int devId)Initializes SHA256 with heap and device ID.
int	**wc_Sha256FinalRaw**(wc_Sha256 * sha256, byte * hash)Gets raw hash without finalizing.
int	**wc_Sha256Transform**(wc_Sha256 * sha, const unsigned char * data)Transforms SHA256 block.
int	**wc_Sha256HashBlock**(wc_Sha256 * sha, const unsigned char * data, unsigned char * hash)Hashes single block and outputs result.
int	**wc_Sha256_Grow**(wc_Sha256 * sha256, const byte * in, int inSz)Grows SHA256 buffer with input data. This function is only available when WOLFSSL_HASH_KEEP is defined. It is used for keeping an internal buffer to hold all data to be hashed rather than iterating over update, which is necessary for some hardware acceleration platforms that have restrictions on streaming hash operations.
int	**wc_Sha256Copy**(wc_Sha256 * src, wc_Sha256 * dst)Copies SHA256 context.
void	**wc_Sha256SizeSet**(wc_Sha256 * sha256, word32 len)Sets SHA256 size.
int	**wc_Sha256SetFlags**(wc_Sha256 * sha256, word32 flags)Sets SHA256 flags.
int	**wc_Sha256GetFlags**(wc_Sha256 * sha256, word32 * flags)Gets SHA256 flags.
int	**wc_InitSha224_ex**(wc_Sha224 * sha224, void * heap, int devId)Initializes SHA224 with heap and device ID.
void	**wc_Sha224Free**(wc_Sha224 * sha224)Frees SHA224 resources.

	Name
int	**wc_Sha224Reset**(wc_Sha224 * sha224)Resets the sha224 structure to its freshly initialized state, continuing with its established allocations, heap hint, and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha224Free for teardown at end of life.
int	**wc_Sha224_Grow**(wc_Sha224 * sha224, const byte * in, int inSz)Grows SHA224 buffer with input data. This function is only available when WOLFSSL_HASH_KEEP is defined. It is used for keeping an internal buffer to hold all data to be hashed rather than iterating over update, which is necessary for some hardware acceleration platforms that have restrictions on streaming hash operations.
int	**wc_Sha224GetHash**(wc_Sha224 * sha224, byte * hash)Gets SHA224 hash without finalizing.
int	**wc_Sha224Copy**(wc_Sha224 * src, wc_Sha224 * dst)Copies SHA224 context.
int	**wc_Sha224SetFlags**(wc_Sha224 * sha224, word32 flags)Sets SHA224 flags.
int	**wc_Sha224GetFlags**(wc_Sha224 * sha224, word32 * flags)Gets SHA224 flags.
int	**wc_InitSha3_224**(wc_Sha3 * sha3, void * heap, int devId)This function initializes SHA3-224. This is automatically called by wc_Sha3_224Hash.
int	**wc_Sha3_224_Update**(wc_Sha3 * sha, const byte * data, word32 len)Can be called to continually hash the provided byte array of length len.
int	**wc_Sha3_224_Final**(wc_Sha3 * sha3, byte * hash)Finalizes hashing of data. Result is placed into hash. Resets state of sha3 struct.
void	**wc_Sha3_224_Free**(wc_Sha3 * sha3)Resets the wc_Sha3 structure. Note: this is only supported if you have WOLFSSL_TI_HASH defined.
int	**wc_Sha3_224_Reset**(wc_Sha3 * sha3)Resets the sha3 structure to its freshly initialized state, continuing with the established heap hint and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha3_224_Free for teardown at end of life.

	Name
int	**wc_Sha3_224_GetHash**(wc_Sha3 * sha3, byte * hash)Gets hash data. Result is placed into hash. Does not reset state of sha3 struct.
int	**wc_Sha3_224_Copy**(wc_Sha3 * sha3, wc_Sha3 * dst)Copy the state of the hash.
int	**wc_InitSha3_256**(wc_Sha3 * sha3, void * heap, int devId)This function initializes SHA3-256. This is automatically called by wc_Sha3_256Hash.
int	**wc_Sha3_256_Update**(wc_Sha3 * sha, const byte * data, word32 len)Can be called to continually hash the provided byte array of length len.
int	**wc_Sha3_256_Final**(wc_Sha3 * sha3, byte * hash)Finalizes hashing of data. Result is placed into hash. Resets state of sha3 struct.
void	**wc_Sha3_256_Free**(wc_Sha3 * sha3)Resets the wc_Sha3 structure. Note: this is only supported if you have WOLFSSL_TI_HASH defined.
int	**wc_Sha3_256_Reset**(wc_Sha3 * sha3)Resets the sha3 structure to its freshly initialized state, continuing with the established heap hint and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha3_256_Free for teardown at end of life.
int	**wc_Sha3_256_GetHash**(wc_Sha3 * sha3, byte * hash)Gets hash data. Result is placed into hash. Does not reset state of sha3 struct.
int	**wc_Sha3_256_Copy**(wc_Sha3 * sha3, wc_Sha3 * dst)Copy the state of the hash.
int	**wc_InitSha3_384**(wc_Sha3 * sha3, void * heap, int devId)This function initializes SHA3-384. This is automatically called by wc_Sha3_384Hash.
int	**wc_Sha3_384_Update**(wc_Sha3 * sha, const byte * data, word32 len)Can be called to continually hash the provided byte array of length len.
int	**wc_Sha3_384_Final**(wc_Sha3 * sha3, byte * hash)Finalizes hashing of data. Result is placed into hash. Resets state of sha3 struct.
void	**wc_Sha3_384_Free**(wc_Sha3 * sha3)Resets the wc_Sha3 structure. Note: this is only supported if you have WOLFSSL_TI_HASH defined.

	Name
int	**wc_Sha3_384_Reset**(wc_Sha3 * sha3)Resets the sha3 structure to its freshly initialized state, continuing with the established heap hint and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha3_384_Free for teardown at end of life.
int	**wc_Sha3_384_GetHash**(wc_Sha3 * sha3, byte * hash)Gets hash data. Result is placed into hash. Does not reset state of sha3 struct.
int	**wc_Sha3_384_Copy**(wc_Sha3 * sha3, wc_Sha3 * dst)Copy the state of the hash.
int	**wc_InitSha3_512**(wc_Sha3 * sha3, void * heap, int devId)This function initializes SHA3-512. This is automatically called by wc_Sha3_512Hash.
int	**wc_Sha3_512_Update**(wc_Sha3 * sha, const byte * data, word32 len)Can be called to continually hash the provided byte array of length len.
int	**wc_Sha3_512_Final**(wc_Sha3 * sha3, byte * hash)Finalizes hashing of data. Result is placed into hash. Resets state of sha3 struct.
void	**wc_Sha3_512_Free**(wc_Sha3 * sha3)Resets the wc_Sha3 structure. Note: this is only supported if you have WOLFSSL_TI_HASH defined.
int	**wc_Sha3_512_Reset**(wc_Sha3 * sha3)Resets the sha3 structure to its freshly initialized state, continuing with the established heap hint and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha3_512_Free for teardown at end of life.
int	**wc_Sha3_512_GetHash**(wc_Sha3 * sha3, byte * hash)Gets hash data. Result is placed into hash. Does not reset state of sha3 struct.
int	**wc_Sha3_512_Copy**(wc_Sha3 * sha3, wc_Sha3 * dst)Copy the state of the hash.
int	**wc_InitShake128**(wc_Shake * shake, void * heap, int devId)This function initializes SHAKE-128. This is automatically called by wc_Shake128Hash.
int	**wc_Shake128_Update**(wc_Shake * sha, const byte * data, word32 len)Can be called to continually hash the provided byte array of length len.

	Name
int	**wc_Shake128_Final**(wc_Shake * shake, byte * hash, word32 hashLen)Finalizes hashing of data. Result is placed into hash. Resets state of shake struct.
int	**wc_Shake128_Absorb**(wc_Shake * sha, const byte * data, word32 len)Called to absorb the provided byte array of length len. Cannot be called incrementally.
int	**wc_Shake128_SqueezeBlocks**(wc_Shake * shake, byte * out, word32 blockCnt)Squeeze out more blocks of data. Result is placed into out. Can be called inrementally.
void	**wc_Shake128_Free**(wc_Shake * shake)Resets the wc_Shake structure. Note: this is only supported if you have WOLFSSL_TI_HASH defined.
int	**wc_Shake128_Reset**(wc_Shake * shake)Resets the shake structure to its freshly initialized state, continuing with the established heap hint and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially absorbed input, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Shake128_Free for teardown at end of life.
int	**wc_Shake128_GetHash**(wc_Shake * shake, byte * hash)Gets hash data. Result is placed into hash. Does not reset state of shake struct.
int	**wc_Shake128_Copy**(wc_Shake * src, wc_Sha3 * dst)Copy the state of the hash.
int	**wc_InitShake256**(wc_Shake * shake, void * heap, int devId)This function initializes SHAKE-256. This is automatically called by wc_Shake256Hash.
int	**wc_Shake256_Update**(wc_Shake * sha, const byte * data, word32 len)Can be called to continually hash the provided byte array of length len.
int	**wc_Shake256_Final**(wc_Shake * shake, byte * hash, word32 hashLen)Finalizes hashing of data. Result is placed into hash. Resets state of shake struct.
int	**wc_Shake256_Absorb**(wc_Shake * sha, const byte * data, word32 len)Called to absorb the provided byte array of length len. Cannot be called incrementally.
int	**wc_Shake256_SqueezeBlocks**(wc_Shake * shake, byte * out, word32 blockCnt)Squeeze out more blocks of data. Result is placed into out. Can be called incrementally.

	Name
void	**wc_Shake256_Free**(wc_Shake * shake)Resets the wc_Shake structure. Note: this is only supported if you have WOLFSSL_TI_HASH defined.
int	**wc_Shake256_Reset**(wc_Shake * shake)Resets the shake structure to its freshly initialized state, continuing with the established heap hint and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially absorbed input, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Shake256_Free for teardown at end of life.
int	**wc_Shake256_GetHash**(wc_Shake * shake, byte * hash)Gets hash data. Result is placed into hash. Does not reset state of shake struct.
int	**wc_Shake256_Copy**(wc_Shake * src, wc_Sha3 * dst)Copy the state of the hash.
int	**wc_InitKmac128**(wc_Kmac * kmac, const byte * key, word32 keyLen, const byte * custom, word32 customLen, void * heap, int devId)This function initializes a KMAC128 operation with a key and an optional customization string, per NIST SP 800_185. KMAC128 requires SHAKE128 (WOLFSSL_SHAKE128) and is enabled with WOLFSSL_KMAC.
int	**wc_Kmac128_Update**(wc_Kmac * kmac, const byte * in, word32 inLen)This function absorbs message data into a KMAC128 operation. It may be called any number of times before finalizing.
int	**wc_Kmac128_Final**(wc_Kmac * kmac, byte * out, word32 outLen)This function finalizes a KMAC128 operation, producing outLen bytes of output. The output length is bound into the result, so a KMAC with a different length is a different value.
int	**wc_Kmac128_FinalXof**(wc_Kmac * kmac, byte * out, word32 outLen)This function finalizes a KMAC128 operation as an XOF (KMACXOF128). Unlike wc_Kmac128_Final, the output length is not bound into the result, so any amount of output may be requested.
void	**wc_Kmac128_Free**(wc_Kmac * kmac)This function releases any resources associated with a KMAC128 operation. Passing NULL is safe and does nothing.

	Name
int	**wc_Kmac128Hash**(const byte * key, word32 keyLen, const byte * custom, word32 customLen, const byte * in, word32 inLen, byte * out, word32 outLen)This function computes a KMAC128 over a single message in one call, initializing, updating and finalizing internally.
int	**wc_Kmac128HashXof**(const byte * key, word32 keyLen, const byte * custom, word32 customLen, const byte * in, word32 inLen, byte * out, word32 outLen)This function computes a KMACXOF128 over a single message in one call. Like wc_Kmac128Hash but the output length is not bound into the result, so any amount of output may be requested.
int	**wc_InitKmac256**(wc_Kmac * kmac, const byte * key, word32 keyLen, const byte * custom, word32 customLen, void * heap, int devId)This function initializes a KMAC256 operation with a key and an optional customization string, per NIST SP 800_185. KMAC256 requires SHAKE256 (WOLFSSL_SHAKE256) and is enabled with WOLFSSL_KMAC.
int	**wc_Kmac256_Update**(wc_Kmac * kmac, const byte * in, word32 inLen)This function absorbs message data into a KMAC256 operation. It may be called any number of times before finalizing.
int	**wc_Kmac256_Final**(wc_Kmac * kmac, byte * out, word32 outLen)This function finalizes a KMAC256 operation, producing outLen bytes of output. The output length is bound into the result, so a KMAC with a different length is a different value.
int	**wc_Kmac256_FinalXof**(wc_Kmac * kmac, byte * out, word32 outLen)This function finalizes a KMAC256 operation as an XOF (KMACXOF256). Unlike wc_Kmac256_Final, the output length is not bound into the result, so any amount of output may be requested.
void	**wc_Kmac256_Free**(wc_Kmac * kmac)This function releases any resources associated with a KMAC256 operation. Passing NULL is safe and does nothing.
int	**wc_Kmac256Hash**(const byte * key, word32 keyLen, const byte * custom, word32 customLen, const byte * in, word32 inLen, byte * out, word32 outLen)This function computes a KMAC256 over a single message in one call, initializing, updating and finalizing internally.

	Name
int	**wc_Kmac256HashXof**(const byte * key, word32 keyLen, const byte * custom, word32 customLen, const byte * in, word32 inLen, byte * out, word32 outLen)This function computes a KMACXOF256 over a single message in one call. Like wc_Kmac256Hash but the output length is not bound into the result, so any amount of output may be requested.
int	**wc_Kmac128_Copy**(wc_Kmac * src, wc_Kmac * dst)This function copies the state of a KMAC128 operation so it can be finalized more than once, for example over a common message prefix.
int	**wc_Kmac256_Copy**(wc_Kmac * src, wc_Kmac * dst)This function copies the state of a KMAC256 operation so it can be finalized more than once, for example over a common message prefix.
int	**wc_InitCshake128**(wc_Cshake * cshake, const byte * name, word32 nameLen, const byte * custom, word32 customLen, void * heap, int devId)This function initializes a cSHAKE128 operation (customizable SHAKE, NIST SP 800_185) with a function_name and customization string. cSHAKE is enabled together with KMAC (WOLFSSL_KMAC). When both strings are empty, cSHAKE reduces to plain SHAKE128.
int	**wc_Cshake128_Update**(wc_Cshake * cshake, const byte * in, word32 inLen)This function absorbs message data into a cSHAKE128 operation.
int	**wc_Cshake128_Final**(wc_Cshake * cshake, byte * out, word32 outLen)This function finalizes a cSHAKE128 operation, squeezing outLen bytes of output. cSHAKE is an XOF, so a longer squeeze extends a shorter one over the same input.
void	**wc_Cshake128_Free**(wc_Cshake * cshake)This function releases any resources associated with a cSHAKE128 operation. Passing NULL is safe and does nothing.
int	**wc_Cshake128**(const byte * name, word32 nameLen, const byte * custom, word32 customLen, const byte * in, word32 inLen, byte * out, word32 outLen)This function computes a cSHAKE128 over a single message in one call, initializing, updating and finalizing internally.

	Name
int	**wc_InitCshake256**(wc_Cshake * cshake, const byte * name, word32 nameLen, const byte * custom, word32 customLen, void * heap, int devId)This function initializes a cSHAKE256 operation (customizable SHAKE, NIST SP 800_185) with a function_name and customization string. cSHAKE is enabled together with KMAC (WOLFSSL_KMAC). When both strings are empty, cSHAKE reduces to plain SHAKE256.
int	**wc_Cshake256_Update**(wc_Cshake * cshake, const byte * in, word32 inLen)This function absorbs message data into a cSHAKE256 operation.
int	**wc_Cshake256_Final**(wc_Cshake * cshake, byte * out, word32 outLen)This function finalizes a cSHAKE256 operation, squeezing outLen bytes of output. cSHAKE is an XOF, so a longer squeeze extends a shorter one over the same input.
void	**wc_Cshake256_Free**(wc_Cshake * cshake)This function releases any resources associated with a cSHAKE256 operation. Passing NULL is safe and does nothing.
int	**wc_Cshake256**(const byte * name, word32 nameLen, const byte * custom, word32 customLen, const byte * in, word32 inLen, byte * out, word32 outLen)This function computes a cSHAKE256 over a single message in one call, initializing, updating and finalizing internally.
int	**wc_Cshake128_Copy**(wc_Cshake * src, wc_Cshake * dst)This function copies the state of a cSHAKE128 operation so it can be finalized more than once, for example over a common message prefix. dst must already be an initialized wc_Cshake.
int	**wc_Cshake256_Copy**(wc_Cshake * src, wc_Cshake * dst)This function copies the state of a cSHAKE256 operation so it can be finalized more than once, for example over a common message prefix. dst must already be an initialized wc_Cshake.
int	**wc_InitSha512**(wc_Sha512 * sha)This function initializes SHA512. This is automatically called by wc_Sha512Hash.
int	**wc_Sha512Update**(wc_Sha512 * sha, const byte * data, word32 len)Can be called to continually hash the provided byte array of length len.
int	**wc_Sha512Final**(wc_Sha512 * sha512, byte * hash)Finalizes hashing of data. Result is placed into hash.

	Name
int	**wc_InitSha384**(wc_Sha384 * sha)This function initializes SHA384. This is automatically called by wc_Sha384Hash.
int	**wc_Sha384Update**(wc_Sha384 * sha, const byte * data, word32 len)Can be called to continually hash the provided byte array of length len.
int	**wc_Sha384Final**(wc_Sha384 * sha384, byte * hash)Finalizes hashing of data. Result is placed into hash.
int	**wc_InitSha512_ex**(wc_Sha512 * sha, void * heap, int devId)Initializes SHA512 with heap and device ID.
int	**wc_Sha512FinalRaw**(wc_Sha512 * sha512, byte * hash)Gets raw hash without finalizing.
void	**wc_Sha512Free**(wc_Sha512 * sha)Frees SHA512 resources.
int	**wc_Sha512Reset**(wc_Sha512 * sha)Resets the sha structure to its freshly initialized state, continuing with its established allocations, heap hint, and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha512Free for teardown at end of life.
int	**wc_Sha512GetHash**(wc_Sha512 * sha512, byte * hash)Gets SHA512 hash without finalizing.
int	**wc_Sha512Copy**(wc_Sha512 * src, wc_Sha512 * dst)Copies SHA512 context.
int	**wc_Sha512_Grow**(wc_Sha512 * sha512, const byte * in, int inSz)Grows SHA512 buffer with input data.
int	**wc_Sha512SetFlags**(wc_Sha512 * sha512, word32 flags)Sets SHA512 flags.
int	**wc_Sha512GetFlags**(wc_Sha512 * sha512, word32 * flags)Gets SHA512 flags.
int	**wc_Sha512Transform**(wc_Sha512 * sha, const unsigned char * data)Transforms SHA512 block.
int	**wc_InitSha512_224**(wc_Sha512 * sha)Initializes SHA512/224.
int	**wc_InitSha512_224_ex**(wc_Sha512 * sha, void * heap, int devId)Initializes SHA512/224 with heap and device ID.
int	**wc_Sha512_224Update**(wc_Sha512 * sha, const byte * data, word32 len)Updates SHA512/224 hash with data.
int	**wc_Sha512_224FinalRaw**(wc_Sha512 * sha512, byte * hash)Gets raw SHA512/224 hash without finalizing.

	Name
int	**wc_Sha512_224Final**(wc_Sha512 * sha512, byte * hash)Finalizes SHA512/224 hash.
void	**wc_Sha512_224Free**(wc_Sha512 * sha)Frees SHA512/224 resources.
int	**wc_Sha512_224Reset**(wc_Sha512 * sha)Resets the sha structure to its freshly initialized state, continuing with its established allocations, heap hint, and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha512_224Free for teardown at end of life.
int	**wc_Sha512_224GetHash**(wc_Sha512 * sha512, byte * hash)Gets SHA512/224 hash without finalizing.
int	**wc_Sha512_224Copy**(wc_Sha512 * src, wc_Sha512 * dst)Copies SHA512/224 context.
int	**wc_Sha512_224SetFlags**(wc_Sha512 * sha512, word32 flags)Sets SHA512/224 flags.
int	**wc_Sha512_224GetFlags**(wc_Sha512 * sha512, word32 * flags)Gets SHA512/224 flags.
int	**wc_Sha512_224Transform**(wc_Sha512 * sha, const unsigned char * data)Transforms SHA512/224 block.
int	**wc_InitSha512_256**(wc_Sha512 * sha)Initializes SHA512/256.
int	**wc_InitSha512_256_ex**(wc_Sha512 * sha, void * heap, int devId)Initializes SHA512/256 with heap and device ID.
int	**wc_Sha512_256Update**(wc_Sha512 * sha, const byte * data, word32 len)Updates SHA512/256 hash with data.
int	**wc_Sha512_256FinalRaw**(wc_Sha512 * sha512, byte * hash)Gets raw SHA512/256 hash without finalizing.
int	**wc_Sha512_256Final**(wc_Sha512 * sha512, byte * hash)Finalizes SHA512/256 hash.
void	**wc_Sha512_256Free**(wc_Sha512 * sha)Frees SHA512/256 resources.
int	**wc_Sha512_256Reset**(wc_Sha512 * sha)Resets the sha structure to its freshly initialized state, continuing with its established allocations, heap hint, and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha512_256Free for teardown at end of life.

	Name
int	**wc_Sha512_256GetHash**(wc_Sha512 * sha512, byte * hash)Gets SHA512/256 hash without finalizing.
int	**wc_Sha512_256Copy**(wc_Sha512 * src, wc_Sha512 * dst)Copies SHA512/256 context.
int	**wc_Sha512_256SetFlags**(wc_Sha512 * sha512, word32 flags)Sets SHA512/256 flags.
int	**wc_Sha512_256GetFlags**(wc_Sha512 * sha512, word32 * flags)Gets SHA512/256 flags.
int	**wc_Sha512_256Transform**(wc_Sha512 * sha, const unsigned char * data)Transforms SHA512/256 block.
int	**wc_InitSha384_ex**(wc_Sha384 * sha, void * heap, int devId)Initializes SHA384 with heap and device ID.
int	**wc_Sha384FinalRaw**(wc_Sha384 * sha384, byte * hash)Gets raw SHA384 hash without finalizing.
void	**wc_Sha384Free**(wc_Sha384 * sha)Frees SHA384 resources.
int	**wc_Sha384Reset**(wc_Sha384 * sha)Resets the sha structure to its freshly initialized state, continuing with its established allocations, heap hint, and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha384Free for teardown at end of life.
int	**wc_Sha384GetHash**(wc_Sha384 * sha384, byte * hash)Gets SHA384 hash without finalizing.
int	**wc_Sha384Copy**(wc_Sha384 * src, wc_Sha384 * dst)Copies SHA384 context.
int	**wc_Sha384_Grow**(wc_Sha384 * sha384, const byte * in, int inSz)Grows SHA384 buffer with input data.
int	**wc_Sha384SetFlags**(wc_Sha384 * sha384, word32 flags)Sets SHA384 flags.
int	**wc_Sha384GetFlags**(wc_Sha384 * sha384, word32 * flags)Gets SHA384 flags.
int	**wc_Sha384Transform**(wc_Sha384 * sha, const unsigned char * data)Transforms SHA384 block.

B.50.2 Functions Documentation

```
int wc_ShaHash(
    const byte * data,
    word32 len,
    byte * hash
)
```

Convenience function, handles all the hashing and places the result into hash.

Parameters:

- **data** the data to hash
- **len** the length of data
- **hash** Byte array to hold hash value.

See:

- wc_ShaHash
- wc_ShaFinal
- wc_InitSha

Return:

- 0 Returned upon successfully ....
- Memory_E memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Example

none

```
int wc_Sha224Hash(
    const byte * data,
    word32 len,
    byte * hash
)
```

Convenience function, handles all the hashing and places the result into hash.

Parameters:

- **data** the data to hash
- **len** the length of data
- **hash** Byte array to hold hash value.

See:

- wc_InitSha224
- wc_Sha224Update
- wc_Sha224Final

Return:

- 0 Success
- <0 Error

Example

none

```
int wc_Sha256Hash(
    const byte * data,
    word32 len,
    byte * hash
)
```

Convenience function, handles all the hashing and places the result into hash.

Parameters:

- **data** the data to hash
- **len** the length of data
- **hash** Byte array to hold hash value.

See:

- wc_Sha256Hash
- wc_Sha256Final
- wc_InitSha256

Return:

- 0 Returned upon successfully ...
- Memory_E memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Example

none

```
int wc_Sha384Hash(
    const byte * data,
    word32 len,
    byte * hash
)
```

Convenience function, handles all the hashing and places the result into hash.

Parameters:

- **data** the data to hash
- **len** the length of data
- **hash** Byte array to hold hash value.

See:

- wc_Sha384Hash
- wc_Sha384Final
- wc_InitSha384

Return:

- 0 Returned upon successfully hashing the data
- Memory_E memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Example

none

```
int wc_Sha512Hash(
    const byte * data,
    word32 len,
    byte * hash
)
```

Convenience function, handles all the hashing and places the result into hash.

Parameters:

- **data** the data to hash
- **len** the length of data
- **hash** Byte array to hold hash value.

See:

- wc_Sha512Hash
- wc_Sha512Final
- wc_InitSha512

Return:

- 0 Returned upon successfully hashing the inputted data
- Memory_E memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Example

none

```
int wc_Sha3_224Hash(
    const byte * data,
    word32 len,
    byte * hash
)
```

Convenience function, handles all the hashing and places the result into hash.

Parameters:

- **data** the data to hash
- **len** the length of data
- **hash** Byte array to hold hash value.

See:

- wc_InitSha3_224
- wc_Sha3_224_Update
- wc_Sha3_224_Final

Return:

- 0 Returned upon successfully hashing the data
- Memory_E memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Example

none

```
int wc_Sha3_256Hash(
    const byte * data,
    word32 len,
    byte * hash
)
```

Convenience function, handles all the hashing and places the result into hash.

Parameters:

- **data** the data to hash
- **len** the length of data
- **hash** Byte array to hold hash value.

See:

- wc_InitSha3_256
- wc_Sha3_256_Update
- wc_Sha3_256_Final

Return:

- 0 Returned upon successfully hashing the data
- Memory_E memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Example

none

```
int wc_Sha3_384Hash(
    const byte * data,
    word32 len,
    byte * hash
)
```

Convenience function, handles all the hashing and places the result into hash.

Parameters:

- **data** the data to hash
- **len** the length of data
- **hash** Byte array to hold hash value.

See:

- wc_InitSha3_384
- wc_Sha3_384_Update
- wc_Sha3_384_Final

Return:

- 0 Returned upon successfully hashing the data
- Memory_E memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Example

none

```
int wc_Sha3_512Hash(
    const byte * data,
    word32 len,
    byte * hash
)
```

Convenience function, handles all the hashing and places the result into hash.

Parameters:

- **data** the data to hash
- **len** the length of data
- **hash** Byte array to hold hash value.

See:

- wc_InitSha3_512
- wc_Sha3_512_Update
- wc_Sha3_512_Final

Return:

- 0 Returned upon successfully hashing the inputted data
- Memory_E memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Example

none

```
int wc_Shake128Hash(
    const byte * data,
    word32 len,
    byte * hash,
    word32 hashLen
)
```

Convenience function, handles all the hashing and places the result into hash.

Parameters:

- **data** the data to hash
- **len** the length of data
- **hash** Byte array to hold hash value.

See:

- wc_InitShake128
- wc_Shake128_Update
- wc_Shake128_Final

Return:

- 0 Returned upon successfully hashing the inputted data

- Memory_E memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Example

none

```
int wc_Shake256Hash(
    const byte * data,
    word32 len,
    byte * hash,
    word32 hashLen
)
```

Convenience function, handles all the hashing and places the result into hash.

Parameters:

- **data** the data to hash
- **len** the length of data
- **hash** Byte array to hold hash value.
- **hashLen** Number of bytes to write to hash.

See:

- wc_InitShake256
- wc_Shake256_Update
- wc_Shake256_Final

Return:

- 0 Returned upon successfully hashing the inputted data
- Memory_E memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Example

none

```
int wc_InitSha(
    wc_Sha * sha
)
```

This function initializes SHA. This is automatically called by wc_ShaHash.

Parameters:

- **sha** pointer to the sha structure to use for encryption

See:

- wc_ShaHash
- wc_ShaUpdate
- wc_ShaFinal

Return: 0 Returned upon successfully initializing

Example

```
Sha sha[1];
if ((ret = wc_InitSha(sha)) != 0) {
   WOLFSSL_MSG("wc_InitSha failed");
}
else {
   wc_ShaUpdate(sha, data, len);
   wc_ShaFinal(sha, hash);
}
```

```
int wc_ShaUpdate(
    wc_Sha * sha,
    const byte * data,
    word32 len
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **sha** pointer to the sha structure to use for encryption
- **data** the data to be hashed
- **len** length of data to be hashed

See:

- wc_ShaHash
- wc_ShaFinal
- wc_InitSha

Return: 0 Returned upon successfully adding the data to the digest.

Example

```
Sha sha[1];
byte data[] = { // Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha(sha)) != 0) {
   WOLFSSL_MSG("wc_InitSha failed");
```

```
}
else {
   wc_ShaUpdate(sha, data, len);
   wc_ShaFinal(sha, hash);
}
```

```
int wc_ShaFinal(
    wc_Sha * sha,
    byte * hash
)
```

Finalizes hashing of data. Result is placed into hash. Resets state of sha struct.

Parameters:

- **sha** pointer to the sha structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_ShaHash
- wc_InitSha
- wc_ShaGetHash

Return: 0 Returned upon successfully finalizing.

Example

```
Sha sha[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha(sha)) != 0) {
   WOLFSSL_MSG("wc_InitSha failed");
}
else {
   wc_ShaUpdate(sha, data, len);
   wc_ShaFinal(sha, hash);
}
```

```
void wc_ShaFree(
    wc_Sha * sha
)
```

Used to clean up memory used by an initialized Sha struct.

Parameters:

- **sha** Pointer to the Sha struct to free.

See:

- wc_InitSha
- wc_ShaUpdate
- wc_ShaFinal

Return: No returns.

Example

```
Sha sha;
wc_InitSha(&sha);
// Use sha
wc_ShaFree(&sha);
```

```
int wc_ShaGetHash(
    wc_Sha * sha,
    byte * hash
)
```

Gets hash data. Result is placed into hash. Does not reset state of sha struct.

Parameters:

- **sha** pointer to the sha structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_ShaHash
- wc_ShaFinal
- wc_InitSha

Return: 0 Returned upon successfully finalizing.

Example

```
Sha sha[1];
if ((ret = wc_InitSha(sha)) != 0) {
WOLFSSL_MSG("wc_InitSha failed");
}
else {
    wc_ShaUpdate(sha, data, len);
    wc_ShaGetHash(sha, hash);
}
```

```
int wc_InitSha_ex(
    wc_Sha * sha,
    void * heap,
    int devId
)
```

Initializes SHA with heap and device ID.

Parameters:

- **sha** SHA structure
- **heap** Heap hint
- **devId** Device ID

See: wc_InitSha

Return:

- 0 on success
- negative on error

Example

```
wc_Sha sha;
int ret = wc_InitSha_ex(&sha, NULL, INVALID_DEVID);
```

```
int wc_ShaFinalRaw(
    wc_Sha * sha,
    byte * hash
)
```

Gets raw hash without finalizing.

Parameters:

- **sha** SHA structure
- **hash** Output hash buffer

See: wc_ShaFinal

Return:

- 0 on success
- negative on error

Example

```
wc_Sha sha;
byte hash[WC_SHA_DIGEST_SIZE];
int ret = wc_ShaFinalRaw(&sha, hash);
```

```
int wc_ShaCopy(
    wc_Sha * src,
    wc_Sha * dst
)
```

Copies SHA context.

Parameters:

- **src** Source SHA structure
- **dst** Destination SHA structure (must be zeroed or previously initialized)

See: wc_InitSha

Return:

- 0 on success
- negative on error

Example

```
wc_Sha src, dst;
memset(&dst, 0, sizeof(dst));
int ret = wc_InitSha(&src);
if (ret == 0)
    ret = wc_ShaCopy(&src, &dst);
```

```
int wc_ShaTransform(
    wc_Sha * sha,
    const unsigned char * data
)
```

Transforms SHA block.

Parameters:

- **sha** SHA structure
- **data** Block data

See: wc_ShaUpdate

Return:

- 0 on success
- negative on error

Example

```
wc_Sha sha;
unsigned char block[WC_SHA_BLOCK_SIZE];
int ret = wc_ShaTransform(&sha, block);
```

```
void wc_ShaSizeSet(
    wc_Sha * sha,
    word32 len
)
```

Sets SHA size.

Parameters:

- **sha** SHA structure
- **len** Size to set

See: wc_ShaUpdate

Return: none No returns

Example

```
wc_Sha sha;
wc_ShaSizeSet(&sha, 1000);
```

```
int wc_ShaSetFlags(
    wc_Sha * sha,
    word32 flags
)
```

Sets SHA flags.

Parameters:

- **sha** SHA structure
- **flags** Flags to set

See: wc_InitSha

Return:

- 0 on success
- negative on error

Example

```
wc_Sha sha;
int ret = wc_ShaSetFlags(&sha, WC_HASH_FLAG_WILLCOPY);
```

```
int wc_InitSha256(
    wc_Sha256 * sha
)
```

This function initializes SHA256. This is automatically called by wc_Sha256Hash.

Parameters:

- **sha256** pointer to the sha256 structure to use for encryption

See:

- wc_Sha256Hash
- wc_Sha256Update
- wc_Sha256Final

Return: 0 Returned upon successfully initializing

Example

```
Sha256 sha256[1];
if ((ret = wc_InitSha256(sha256)) != 0) {
    WOLFSSL_MSG("wc_InitSha256 failed");
}
else {
    wc_Sha256Update(sha256, data, len);
    wc_Sha256Final(sha256, hash);
}
```

```
int wc_Sha256Update(
    wc_Sha256 * sha,
    const byte * data,
    word32 len
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **sha256** pointer to the sha256 structure to use for encryption
- **data** the data to be hashed
- **len** length of data to be hashed

See:

- wc_Sha256Hash
- wc_Sha256Final
- wc_InitSha256

Return: 0 Returned upon successfully adding the data to the digest.

Example

```
Sha256 sha256[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha256(sha256)) != 0) {
   WOLFSSL_MSG("wc_InitSha256 failed");
}
else {
    wc_Sha256Update(sha256, data, len);
    wc_Sha256Final(sha256, hash);
}
```

```
int wc_Sha256Final(
    wc_Sha256 * sha256,
    byte * hash
)
```

Finalizes hashing of data. Result is placed into hash. Resets state of sha256 struct.

Parameters:

- **sha256** pointer to the sha256 structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Sha256Hash
- wc_Sha256GetHash
- wc_InitSha256

Return: 0 Returned upon successfully finalizing.

Example

```
Sha256 sha256[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha256(sha256)) != 0) {
   WOLFSSL_MSG("wc_InitSha256 failed");
}
else {
   wc_Sha256Update(sha256, data, len);
   wc_Sha256Final(sha256, hash);
}
```

```
void wc_Sha256Free(
    wc_Sha256 * sha256
)
```

Resets the Sha256 structure. Note: this is only supported if you have WOLFSSL_TI_HASH defined.

Parameters:

- **sha256** Pointer to the sha256 structure to be freed.

See:

- wc_InitSha256
- wc_Sha256Update
- wc_Sha256Final

Return: none No returns.

Example

```
Sha256 sha256;
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha256(&sha256)) != 0) {
    WOLFSSL_MSG("wc_InitSha256 failed");
}
else {
    wc_Sha256Update(&sha256, data, len);
    wc_Sha256Final(&sha256, hash);
    wc_Sha256Free(&sha256);
}
```

```
int wc_Sha256Reset(
    wc_Sha256 * sha256
)
```

Resets the sha256 structure to its freshly initialized state, continuing with its established allocations, heap hint, and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha256Free for teardown at end of life.

Parameters:

- **sha256** Pointer to the sha256 structure to be reset.

See:

- wc_InitSha256

- wc_Sha256Final
- wc_Sha256Free

Return:

- 0 Returned upon successfully resetting the structure.
- BAD_FUNC_ARG Returned if sha256 is NULL.
- negative Other values may be returned on reinitialization failure in fallback (non-plain-software) configurations.

Example

```
wc_Sha256 sha256;
byte data[] = { Data to be discarded };
byte data2[] = { Data to be hashed };
byte hash[WC_SHA256_DIGEST_SIZE];

if ((ret = wc_InitSha256(&sha256)) == 0) {
    wc_Sha256Update(&sha256, data, sizeof(data));
    wc_Sha256Reset(&sha256); /* discard the partially hashed message */
    wc_Sha256Update(&sha256, data2, sizeof(data2));
    wc_Sha256Final(&sha256, hash);
    wc_Sha256Free(&sha256);
}
```

```
int wc_Sha256GetHash(
    wc_Sha256 * sha256,
    byte * hash
)
```

Gets hash data. Result is placed into hash. Does not reset state of sha256 struct.

Parameters:

- **sha256** pointer to the sha256 structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Sha256Hash
- wc_Sha256Final
- wc_InitSha256

Return: 0 Returned upon successfully finalizing.

Example

```
Sha256 sha256[1];
if ((ret = wc_InitSha256(sha256)) != 0) {
   WOLFSSL_MSG("wc_InitSha256 failed");
```

```
}
else {
   wc_Sha256Update(sha256, data, len);
   wc_Sha256GetHash(sha256, hash);
}
```

```
int wc_InitSha224(
    wc_Sha224 * sha224
)
```

Used to initialize a Sha224 struct.

Parameters:

- **sha224** Pointer to a Sha224 struct to initialize.

See:

- wc_Sha224Hash
- wc_Sha224Update
- wc_Sha224Final

Return:

- 0 Success
- 1 Error returned because sha224 is null.

Example

```
Sha224 sha224;
if(wc_InitSha224(&sha224) != 0)
{
    // Handle error
}
```

```
int wc_Sha224Update(
    wc_Sha224 * sha224,
    const byte * data,
    word32 len
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **sha224** Pointer to the Sha224 structure to use for encryption.
- **data** Data to be hashed.
- **len** Length of data to be hashed.

See:

- wc_InitSha224
- wc_Sha224Final
- wc_Sha224Hash

Return:

- 0 Success
- 1 Error returned if function fails.
- BAD_FUNC_ARG Error returned if sha224 or data is null.

Example

```
Sha224 sha224;
byte data[]; // Data to be hashed
word32 len = sizeof(data);

if ((ret = wc_InitSha224(&sha224)) != 0) {
   WOLFSSL_MSG("wc_InitSha224 failed");
}
else {
  wc_Sha224Update(&sha224, data, len);
  wc_Sha224Final(&sha224, hash);
}
```

```
int wc_Sha224Final(
    wc_Sha224 * sha224,
    byte * hash
)
```

Finalizes hashing of data. Result is placed into hash. Resets state of sha224 struct.

Parameters:

- **sha224** pointer to the sha224 structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_InitSha224
- wc_Sha224Hash
- wc_Sha224Update

Return:

- 0 Success
- <0 Error

Example

```
Sha224 sha224;
byte data[]; // Data to be hashed
word32 len = sizeof(data);

if ((ret = wc_InitSha224(&sha224)) != 0) {
    WOLFSSL_MSG("wc_InitSha224 failed");
}
else {
    wc_Sha224Update(&sha224, data, len);
    wc_Sha224Final(&sha224, hash);
}
```

```
int wc_InitSha256_ex(
    wc_Sha256 * sha,
    void * heap,
    int devId
)
```

Initializes SHA256 with heap and device ID.

Parameters:

- **sha** SHA256 structure
- **heap** Heap hint
- **devId** Device ID

See: wc_InitSha256

Return:

- 0 on success
- negative on error

Example

```
wc_Sha256 sha;
int ret = wc_InitSha256_ex(&sha, NULL, INVALID_DEVID);
```

```
int wc_Sha256FinalRaw(
    wc_Sha256 * sha256,
    byte * hash
)
```

Gets raw hash without finalizing.

Parameters:

- **sha256** SHA256 structure
- **hash** Output hash buffer

See: wc_Sha256Final

Return:

- 0 on success
- negative on error

Example

```
wc_Sha256 sha;
byte hash[WC_SHA256_DIGEST_SIZE];
int ret = wc_Sha256FinalRaw(&sha, hash);
```

```
int wc_Sha256Transform(
    wc_Sha256 * sha,
    const unsigned char * data
)
```

Transforms SHA256 block.

Parameters:

- **sha** SHA256 structure
- **data** Block data

See: wc_Sha256Update

Return:

- 0 on success
- negative on error

Example

```
wc_Sha256 sha;
unsigned char block[WC_SHA256_BLOCK_SIZE];
int ret = wc_Sha256Transform(&sha, block);
```

```
int wc_Sha256HashBlock(
    wc_Sha256 * sha,
    const unsigned char * data,
    unsigned char * hash
)
```

Hashes single block and outputs result.

Parameters:

- **sha** SHA256 structure
- **data** Block data
- **hash** Output hash buffer

See: wc_Sha256Transform

Return:

- 0 on success
- negative on error

Example

```
wc_Sha256 sha;
unsigned char block[WC_SHA256_BLOCK_SIZE];
unsigned char hash[WC_SHA256_DIGEST_SIZE];
int ret = wc_Sha256HashBlock(&sha, block, hash);
```

```
int wc_Sha256_Grow(
    wc_Sha256 * sha256,
    const byte * in,
    int inSz
)
```

Grows SHA256 buffer with input data. This function is only available when WOLFSSL_HASH_KEEP is defined. It is used for keeping an internal buffer to hold all data to be hashed rather than iterating over update, which is necessary for some hardware acceleration platforms that have restrictions on streaming hash operations.

Parameters:

- **sha256** SHA256 structure
- **in** Input data
- **inSz** Input size

See: wc_Sha256Update

Return:

- 0 on success
- negative on error

Example

```
wc_Sha256 sha;
byte data[100];
int ret = wc_Sha256_Grow(&sha, data, sizeof(data));
```

```
int wc_Sha256Copy(
    wc_Sha256 * src,
    wc_Sha256 * dst
)
```

Copies SHA256 context.

Parameters:

- **src** Source SHA256 structure
- **dst** Destination SHA256 structure; (must be zeroed or previously initialized)

See: wc_InitSha256

Return:

- 0 on success
- negative on error

Example

```
wc_Sha256 src, dst;
memset(&dst, 0, sizeof(dst));
int ret = wc_InitSha256(&src);
if (ret == 0)
    ret = wc_Sha256Copy(&src, &dst);
```

```
void wc_Sha256SizeSet(
    wc_Sha256 * sha256,
    word32 len
)
```

Sets SHA256 size.

Parameters:

- **sha256** SHA256 structure
- **len** Size to set

See: wc_Sha256Update

Return: none No returns

Example

```
wc_Sha256 sha;
wc_Sha256SizeSet(&sha, 1000);
```

```
int wc_Sha256SetFlags(
    wc_Sha256 * sha256,
    word32 flags
)
```

Sets SHA256 flags.

Parameters:

- **sha256** SHA256 structure
- **flags** Flags to set

See: wc_InitSha256

Return:

- 0 on success
- negative on error

Example

```
wc_Sha256 sha;
int ret = wc_Sha256SetFlags(&sha, WC_HASH_FLAG_WILLCOPY);
```

```
int wc_Sha256GetFlags(
    wc_Sha256 * sha256,
    word32 * flags
)
```

Gets SHA256 flags.

Parameters:

- **sha256** SHA256 structure
- **flags** Pointer to store flags

See: wc_Sha256SetFlags

Return:

- 0 on success
- negative on error

Example

```
wc_Sha256 sha;
word32 flags;
int ret = wc_Sha256GetFlags(&sha, &flags);
```

```
int wc_InitSha224_ex(
    wc_Sha224 * sha224,
    void * heap,
    int devId
)
```

Initializes SHA224 with heap and device ID.

Parameters:

- **sha224** SHA224 structure
- **heap** Heap hint
- **devId** Device ID

See: wc_InitSha224

Return:

- 0 on success
- negative on error

Example

```
wc_Sha224 sha;
int ret = wc_InitSha224_ex(&sha, NULL, INVALID_DEVID);
```

```
void wc_Sha224Free(
    wc_Sha224 * sha224
)
```

Frees SHA224 resources.

Parameters:

- **sha224** SHA224 structure

See: wc_InitSha224

Return: none No returns

Example

```
wc_Sha224 sha;
wc_InitSha224(&sha);
wc_Sha224Free(&sha);
```

```
int wc_Sha224Reset(
    wc_Sha224 * sha224
)
```

Resets the sha224 structure to its freshly initialized state, continuing with its established allocations, heap hint, and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha224Free for teardown at end of life.

Parameters:

- **sha224** Pointer to the sha224 structure to be reset.

See:

- wc_InitSha224
- wc_Sha224Final
- wc_Sha224Free

Return:

- 0 Returned upon successfully resetting the structure.
- BAD_FUNC_ARG Returned if sha224 is NULL.
- negative Other values may be returned on reinitialization failure in fallback (non-plain-software) configurations.

Example

```
wc_Sha224 sha224;
byte data[] = { Data to be discarded };
byte data2[] = { Data to be hashed };
byte hash[WC_SHA224_DIGEST_SIZE];

if ((ret = wc_InitSha224(&sha224)) == 0) {
    wc_Sha224Update(&sha224, data, sizeof(data));
    wc_Sha224Reset(&sha224); /* discard the partially hashed message */
    wc_Sha224Update(&sha224, data2, sizeof(data2));
    wc_Sha224Final(&sha224, hash);
    wc_Sha224Free(&sha224);
}
```

```
int wc_Sha224_Grow(
    wc_Sha224 * sha224,
    const byte * in,
    int inSz
)
```

Grows SHA224 buffer with input data. This function is only available when WOLFSSL_HASH_KEEP is defined. It is used for keeping an internal buffer to hold all data to be hashed rather than iterating

over update, which is necessary for some hardware acceleration platforms that have restrictions on streaming hash operations.

Parameters:

- **sha224** SHA224 structure
- **in** Input data
- **inSz** Input size

See: wc_Sha224Update

Return:

- 0 on success
- negative on error

Example

```
wc_Sha224 sha;
byte data[100];
int ret = wc_Sha224_Grow(&sha, data, sizeof(data));
```

```
int wc_Sha224GetHash(
    wc_Sha224 * sha224,
    byte * hash
)
```

Gets SHA224 hash without finalizing.

Parameters:

- **sha224** SHA224 structure
- **hash** Output hash buffer

See: wc_Sha224Final

Return:

- 0 on success
- negative on error

Example

```
wc_Sha224 sha;
byte hash[WC_SHA224_DIGEST_SIZE];
int ret = wc_Sha224GetHash(&sha, hash);
```

```
int wc_Sha224Copy(
    wc_Sha224 * src,
    wc_Sha224 * dst
)
```

Copies SHA224 context.

Parameters:

- **src** Source SHA224 structure
- **dst** Destination SHA224 structure (must be zeroed or previously initialized)

See: wc_InitSha224

Return:

- 0 on success
- negative on error

Example

```
wc_Sha224 src, dst;
memset(&dst, 0, sizeof(dst));
int ret = wc_InitSha224(&src);
if (ret == 0)
    ret = wc_Sha224Copy(&src, &dst);
```

```
int wc_Sha224SetFlags(
    wc_Sha224 * sha224,
    word32 flags
)
```

Sets SHA224 flags.

Parameters:

- **sha224** SHA224 structure
- **flags** Flags to set

See: wc_InitSha224

Return:

- 0 on success
- negative on error

Example

```
wc_Sha224 sha;
int ret = wc_Sha224SetFlags(&sha, WC_HASH_FLAG_WILLCOPY);
```

```
int wc_Sha224GetFlags(
    wc_Sha224 * sha224,
    word32 * flags
)
```

Gets SHA224 flags.

Parameters:

- **sha224** SHA224 structure
- **flags** Pointer to store flags

See: wc_Sha224SetFlags

Return:

- 0 on success
- negative on error

Example

```
wc_Sha224 sha;
word32 flags;
int ret = wc_Sha224GetFlags(&sha, &flags);
```

```
int wc_InitSha3_224(
    wc_Sha3 * sha3,
    void * heap,
    int devId
)
```

This function initializes SHA3-224. This is automatically called by wc_Sha3_224Hash.

Parameters:

- **sha3** pointer to the sha3 structure to use for encryption

See:

- wc_Sha3_224Hash
- wc_Sha3_224_Update
- wc_Sha3_224_Final

Return: 0 Returned upon successfully initializing

Example

```
wc_Sha3 sha3[1];
if ((ret = wc_InitSha3_224(sha3, NULL, INVALID_DEVID)) != 0) {
    WOLFSSL_MSG("wc_InitSha3_224 failed");
}
else {
    wc_Sha3_224_Update(sha3, data, len);
    wc_Sha3_224_Final(sha3, hash);
}
```

```
int wc_Sha3_224_Update(
    wc_Sha3 * sha,
    const byte * data,
    word32 len
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **sha3** pointer to the sha3 structure to use for encryption
- **data** the data to be hashed
- **len** length of data to be hashed

See:

- wc_Sha3_224Hash
- wc_Sha3_224_Final
- wc_InitSha3_224

Return: 0 Returned upon successfully adding the data to the digest.

Example

```
wc_Sha3 sha3[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha3_224(sha3, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitSha3_224 failed");
}
else {
    wc_Sha3_224_Update(sha3, data, len);
    wc_Sha3_224_Final(sha3, hash);
}
```

```
int wc_Sha3_224_Final(
    wc_Sha3 * sha3,
    byte * hash
)
```

Finalizes hashing of data. Result is placed into hash. Resets state of sha3 struct.

Parameters:

- **sha3** pointer to the sha3 structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Sha3_224Hash
- wc_Sha3_224_GetHash
- wc_InitSha3_224

Return: 0 Returned upon successfully finalizing.

Example

```
wc_Sha3 sha3[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha3_224(sha3, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitSha3_224 failed");
}
else {
   wc_Sha3_224_Update(sha3, data, len);
   wc_Sha3_224_Final(sha3, hash);
}
```

```
void wc_Sha3_224_Free(
    wc_Sha3 * sha3
)
```

Resets the wc_Sha3 structure. Note: this is only supported if you have WOLFSSL_TI_HASH defined.

Parameters:

- **sha3** Pointer to the sha3 structure to be freed.

See:

- wc_InitSha3_224
- wc_Sha3_224_Update
- wc_Sha3_224_Final

Return: none No returns.

Example

```
wc_Sha3 sha3;
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha3_224(&sha3, NULL, INVALID_DEVID)) != 0) {
    WOLFSSL_MSG("wc_InitSha3_224 failed");
}
else {
    wc_Sha3_224_Update(&sha3, data, len);
    wc_Sha3_224_Final(&sha3, hash);
    wc_Sha3_224_Free(&sha3);
}
```

```
int wc_Sha3_224_Reset(
    wc_Sha3 * sha3
)
```

Resets the sha3 structure to its freshly initialized state, continuing with the established heap hint and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha3_224_Free for teardown at end of life.

Parameters:

- **sha3** Pointer to the sha3 structure to be reset.

See:

- wc_InitSha3_224
- wc_Sha3_224_Final
- wc_Sha3_224_Free

Return:

- 0 Returned upon successfully resetting the structure.
- BAD_FUNC_ARG Returned if sha3 is NULL.
- negative Other values may be returned on reinitialization failure in fallback (non-plain-software) configurations.

Example

```
wc_Sha3 sha3;
byte data[] = { Data to be discarded };
byte data2[] = { Data to be hashed };
byte hash[WC_SHA3_224_DIGEST_SIZE];

if ((ret = wc_InitSha3_224(&sha3, NULL, INVALID_DEVID)) == 0) {
    wc_Sha3_224_Update(&sha3, data, sizeof(data));
    wc_Sha3_224_Reset(&sha3); /* discard the partially hashed message */
    wc_Sha3_224_Update(&sha3, data2, sizeof(data2));
```

```
    wc_Sha3_224_Final(&sha3, hash);
    wc_Sha3_224_Free(&sha3);
}
```

```
int wc_Sha3_224_GetHash(
    wc_Sha3 * sha3,
    byte * hash
)
```

Gets hash data. Result is placed into hash. Does not reset state of sha3 struct.

Parameters:

- **sha3** pointer to the sha3 structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Sha3_224Hash
- wc_Sha3_224_Final
- wc_InitSha3_224
- wc_Sha3_224_Copy

Return: 0 Returned upon successful copying of the hash.

Example

```
wc_Sha3 sha3[1];
if ((ret = wc_InitSha3_224(sha3, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitSha3_224 failed");
}
else {
   wc_Sha3_224_Update(sha3, data, len);
   wc_Sha3_224_GetHash(sha3, hash);
}
```

```
int wc_Sha3_224_Copy(
    wc_Sha3 * sha3,
    wc_Sha3 * dst
)
```

Copy the state of the hash.

Parameters:

- **sha3** pointer to the sha3 structure to copy
- **dst** pointer to the sha3 structure to copy into (must be zeroed or previously initialized)

See:

- wc_Sha3_224Hash
- wc_Sha3_224_Final
- wc_InitSha3_224
- wc_Sha3_224_GetHash

Return: 0 Returned upon successful copying.

Example

```
wc_Sha3 sha3[1];
wc_Sha3 sha3_dup[1];
memset(sha3_dup, 0, sizeof(*sha3_dup));
if ((ret = wc_InitSha3_224(sha3, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitSha3_224 failed");
}
else {
   wc_Sha3_224_Update(sha3, data, len);
   wc_Sha3_224_Copy(sha3, sha3_dup);
}
```

```
int wc_InitSha3_256(
    wc_Sha3 * sha3,
    void * heap,
    int devId
)
```

This function initializes SHA3-256. This is automatically called by wc_Sha3_256Hash.

Parameters:

- **sha3** pointer to the sha3 structure to use for encryption

See:

- wc_Sha3_256Hash
- wc_Sha3_256_Update
- wc_Sha3_256_Final

Return: 0 Returned upon successfully initializing

Example

```
wc_Sha3 sha3[1];
if ((ret = wc_InitSha3_256(sha3, NULL, INVALID_DEVID)) != 0) {
    WOLFSSL_MSG("wc_InitSha3_256 failed");
}
else {
    wc_Sha3_256_Update(sha3, data, len);
```

```
    wc_Sha3_256_Final(sha3, hash);
}



int wc_Sha3_256_Update(
    wc_Sha3 * sha,
    const byte * data,
    word32 len
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **sha3** pointer to the sha3 structure to use for encryption
- **data** the data to be hashed
- **len** length of data to be hashed

See:

- wc_Sha3_256Hash
- wc_Sha3_256_Final
- wc_InitSha3_256

Return: 0 Returned upon successfully adding the data to the digest.

Example

```
wc_Sha3 sha3[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha3_256(sha3, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitSha3_256 failed");
}
else {
    wc_Sha3_256_Update(sha3, data, len);
    wc_Sha3_256_Final(sha3, hash);
}



int wc_Sha3_256_Final(
    wc_Sha3 * sha3,
    byte * hash
)
```

Finalizes hashing of data. Result is placed into hash. Resets state of sha3 struct.

Parameters:

- **sha3** pointer to the sha3 structure to use for encryption

- **hash** Byte array to hold hash value.

See:

- wc_Sha3_256Hash
- wc_Sha3_256_GetHash
- wc_InitSha3_256

Return: 0 Returned upon successfully finalizing.

Example

```
wc_Sha3 sha3[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha3_256(sha3, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitSha3_256 failed");
}
else {
   wc_Sha3_256_Update(sha3, data, len);
   wc_Sha3_256_Final(sha3, hash);
}
```

```
void wc_Sha3_256_Free(
    wc_Sha3 * sha3
)
```

Resets the wc_Sha3 structure. Note: this is only supported if you have WOLFSSL_TI_HASH defined.

Parameters:

- **sha3** Pointer to the sha3 structure to be freed.

See:

- wc_InitSha3_256
- wc_Sha3_256_Update
- wc_Sha3_256_Final

Return: none No returns.

Example

```
wc_Sha3 sha3;
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha3_256(&sha3, NULL, INVALID_DEVID)) != 0) {
    WOLFSSL_MSG("wc_InitSha3_256 failed");
}
```

```
else {
    wc_Sha3_256_Update(&sha3, data, len);
    wc_Sha3_256_Final(&sha3, hash);
    wc_Sha3_256_Free(&sha3);
}
```

```
int wc_Sha3_256_Reset(
    wc_Sha3 * sha3
)
```

Resets the sha3 structure to its freshly initialized state, continuing with the established heap hint and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha3_256_Free for teardown at end of life.

Parameters:

- **sha3** Pointer to the sha3 structure to be reset.

See:

- wc_InitSha3_256
- wc_Sha3_256_Final
- wc_Sha3_256_Free

Return:

- 0 Returned upon successfully resetting the structure.
- BAD_FUNC_ARG Returned if sha3 is NULL.
- negative Other values may be returned on reinitialization failure in fallback (non-plain-software) configurations.

Example

```
wc_Sha3 sha3;
byte data[] = { Data to be discarded };
byte data2[] = { Data to be hashed };
byte hash[WC_SHA3_256_DIGEST_SIZE];

if ((ret = wc_InitSha3_256(&sha3, NULL, INVALID_DEVID)) == 0) {
    wc_Sha3_256_Update(&sha3, data, sizeof(data));
    wc_Sha3_256_Reset(&sha3); /* discard the partially hashed message */
    wc_Sha3_256_Update(&sha3, data2, sizeof(data2));
    wc_Sha3_256_Final(&sha3, hash);
    wc_Sha3_256_Free(&sha3);
}
```

```
int wc_Sha3_256_GetHash(
    wc_Sha3 * sha3,
    byte * hash
)
```

Gets hash data. Result is placed into hash. Does not reset state of sha3 struct.

Parameters:

- **sha3** pointer to the sha3 structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Sha3_256Hash
- wc_Sha3_256_Final
- wc_InitSha3_256
- wc_Sha3_256_Copy

Return: 0 Returned upon successful copying of the hash.

Example

```
wc_Sha3 sha3[1];
if ((ret = wc_InitSha3_256(sha3, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitSha3_256 failed");
}
else {
   wc_Sha3_256_Update(sha3, data, len);
   wc_Sha3_256_GetHash(sha3, hash);
}
```

```
int wc_Sha3_256_Copy(
    wc_Sha3 * sha3,
    wc_Sha3 * dst
)
```

Copy the state of the hash.

Parameters:

- **sha3** pointer to the sha3 structure to copy
- **dst** pointer to the sha3 structure to copy into (must be zeroed or previously initialized)

See:

- wc_Sha3_256Hash
- wc_Sha3_256_Final
- wc_InitSha3_256
- wc_Sha3_256_GetHash

Return: 0 Returned upon successful copying.

Example

```
wc_Sha3 sha3[1];
wc_Sha3 sha3_dup[1];
memset(sha3_dup, 0, sizeof(*sha3_dup));
if ((ret = wc_InitSha3_256(sha3, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitSha3_256 failed");
}
else {
   wc_Sha3_256_Update(sha3, data, len);
   wc_Sha3_256_Copy(sha3, sha3_dup);
}
```

```
int wc_InitSha3_384(
    wc_Sha3 * sha3,
    void * heap,
    int devId
)
```

This function initializes SHA3-384. This is automatically called by wc_Sha3_384Hash.

Parameters:

- **sha3** pointer to the sha3 structure to use for encryption

See:

- wc_Sha3_384Hash
- wc_Sha3_384_Update
- wc_Sha3_384_Final

Return: 0 Returned upon successfully initializing

Example

```
wc_Sha3 sha3[1];
if ((ret = wc_InitSha3_384(sha3, NULL, INVALID_DEVID)) != 0) {
    WOLFSSL_MSG("wc_InitSha3_384 failed");
}
else {
    wc_Sha3_384_Update(sha3, data, len);
    wc_Sha3_384_Final(sha3, hash);
}
```

```
int wc_Sha3_384_Update(
    wc_Sha3 * sha,
    const byte * data,
    word32 len
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **sha3** pointer to the sha3 structure to use for encryption
- **data** the data to be hashed
- **len** length of data to be hashed

See:

- wc_Sha3_384Hash
- wc_Sha3_384_Final
- wc_InitSha3_384

Return: 0 Returned upon successfully adding the data to the digest.

Example

```
wc_Sha3 sha3[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha3_384(sha3, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitSha3_384 failed");
}
else {
    wc_Sha3_384_Update(sha3, data, len);
    wc_Sha3_384_Final(sha3, hash);
}
```

```
int wc_Sha3_384_Final(
    wc_Sha3 * sha3,
    byte * hash
)
```

Finalizes hashing of data. Result is placed into hash. Resets state of sha3 struct.

Parameters:

- **sha3** pointer to the sha3 structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Sha3_384Hash
- wc_Sha3_384_GetHash
- wc_InitSha3_384

Return: 0 Returned upon successfully finalizing.

Example

```
wc_Sha3 sha3[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha3_384(sha3, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitSha3_384 failed");
}
else {
   wc_Sha3_384_Update(sha3, data, len);
   wc_Sha3_384_Final(sha3, hash);
}
```

```
void wc_Sha3_384_Free(
    wc_Sha3 * sha3
)
```

Resets the wc_Sha3 structure. Note: this is only supported if you have WOLFSSL_TI_HASH defined.

Parameters:

- **sha3** Pointer to the sha3 structure to be freed.

See:

- wc_InitSha3_384
- wc_Sha3_384_Update
- wc_Sha3_384_Final

Return: none No returns.

Example

```
wc_Sha3 sha3;
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha3_384(&sha3, NULL, INVALID_DEVID)) != 0) {
    WOLFSSL_MSG("wc_InitSha3_384 failed");
}
else {
    wc_Sha3_384_Update(&sha3, data, len);
    wc_Sha3_384_Final(&sha3, hash);
```

```
    wc_Sha3_384_Free(&sha3);
}



int wc_Sha3_384_Reset(
    wc_Sha3 * sha3
)
```

Resets the sha3 structure to its freshly initialized state, continuing with the established heap hint and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha3_384_Free for teardown at end of life.

Parameters:

- **sha3** Pointer to the sha3 structure to be reset.

See:

- wc_InitSha3_384
- wc_Sha3_384_Final
- wc_Sha3_384_Free

Return:

- 0 Returned upon successfully resetting the structure.
- BAD_FUNC_ARG Returned if sha3 is NULL.
- negative Other values may be returned on reinitialization failure in fallback (non-plain-software) configurations.

Example

```
wc_Sha3 sha3;
byte data[] = { Data to be discarded };
byte data2[] = { Data to be hashed };
byte hash[WC_SHA3_384_DIGEST_SIZE];

if ((ret = wc_InitSha3_384(&sha3, NULL, INVALID_DEVID)) == 0) {
    wc_Sha3_384_Update(&sha3, data, sizeof(data));
    wc_Sha3_384_Reset(&sha3); /* discard the partially hashed message */
    wc_Sha3_384_Update(&sha3, data2, sizeof(data2));
    wc_Sha3_384_Final(&sha3, hash);
    wc_Sha3_384_Free(&sha3);
}



int wc_Sha3_384_GetHash(
    wc_Sha3 * sha3,
    byte * hash
)
```

Gets hash data. Result is placed into hash. Does not reset state of sha3 struct.

Parameters:

- **sha3** pointer to the sha3 structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Sha3_384Hash
- wc_Sha3_384_Final
- wc_InitSha3_384
- wc_Sha3_384_Copy

Return: 0 Returned upon successful copying of the hash.

Example

```
wc_Sha3 sha3[1];
if ((ret = wc_InitSha3_384(sha3, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitSha3_38384ailed");
}
else {
   wc_Sha3_384_Update(sha3, data, len);
   wc_Sha3_384_GetHash(sha3, hash);
}
```

```
int wc_Sha3_384_Copy(
    wc_Sha3 * sha3,
    wc_Sha3 * dst
)
```

Copy the state of the hash.

Parameters:

- **sha3** pointer to the sha3 structure to copy
- **dst** pointer to the sha3 structure to copy into (must be zeroed or previously initialized)

See:

- wc_Sha3_384Hash
- wc_Sha3_384_Final
- wc_InitSha3_384
- wc_Sha3_384_GetHash

Return: 0 Returned upon successful copying.

Example

```
wc_Sha3 sha3[1];
wc_Sha3 sha3_dup[1];
memset(sha3_dup, 0, sizeof(*sha3_dup));
if ((ret = wc_InitSha3_384(sha3, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitSha3_384 failed");
}
else {
   wc_Sha3_384_Update(sha3, data, len);
   wc_Sha3_384_Copy(sha3, sha3_dup);
}
```

```
int wc_InitSha3_512(
    wc_Sha3 * sha3,
    void * heap,
    int devId
)
```

This function initializes SHA3-512. This is automatically called by wc_Sha3_512Hash.

Parameters:

- **sha3** pointer to the sha3 structure to use for encryption

See:

- wc_Sha3_512Hash
- wc_Sha3_512_Update
- wc_Sha3_512_Final

Return: 0 Returned upon successfully initializing

Example

```
wc_Sha3 sha3[1];
if ((ret = wc_InitSha3_512(sha3, NULL, INVALID_DEVID)) != 0) {
    WOLFSSL_MSG("wc_InitSha3_512 failed");
}
else {
    wc_Sha3_512_Update(sha3, data, len);
    wc_Sha3_512_Final(sha3, hash);
}
```

```
int wc_Sha3_512_Update(
    wc_Sha3 * sha,
    const byte * data,
    word32 len
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **sha3** pointer to the sha3 structure to use for encryption
- **data** the data to be hashed
- **len** length of data to be hashed

See:

- wc_Sha3_512Hash
- wc_Sha3_512_Final
- wc_InitSha3_512

Return: 0 Returned upon successfully adding the data to the digest.

Example

```
wc_Sha3 sha3[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha3_512(sha3, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitSha3_512 failed");
}
else {
    wc_Sha3_512_Update(sha3, data, len);
    wc_Sha3_512_Final(sha3, hash);
}
```

```
int wc_Sha3_512_Final(
    wc_Sha3 * sha3,
    byte * hash
)
```

Finalizes hashing of data. Result is placed into hash. Resets state of sha3 struct.

Parameters:

- **sha3** pointer to the sha3 structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Sha3_512Hash
- wc_Sha3_512_GetHash
- wc_InitSha3_512

Return: 0 Returned upon successfully finalizing.

Example

```
wc_Sha3 sha3[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha3_512(sha3, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitSha3_512 failed");
}
else {
   wc_Sha3_512_Update(sha3, data, len);
   wc_Sha3_512_Final(sha3, hash);
}
```

```
void wc_Sha3_512_Free(
    wc_Sha3 * sha3
)
```

Resets the wc_Sha3 structure. Note: this is only supported if you have WOLFSSL_TI_HASH defined.

Parameters:

- **sha3** Pointer to the sha3 structure to be freed.

See:

- wc_InitSha3_512
- wc_Sha3_512_Update
- wc_Sha3_512_Final

Return: none No returns.

Example

```
wc_Sha3 sha3;
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha3_512(&sha3, NULL, INVALID_DEVID)) != 0) {
    WOLFSSL_MSG("wc_InitSha3_512 failed");
}
else {
    wc_Sha3_512_Update(&sha3, data, len);
    wc_Sha3_512_Final(&sha3, hash);
    wc_Sha3_512_Free(&sha3);
}
```

```
int wc_Sha3_512_Reset(
    wc_Sha3 * sha3
)
```

Resets the sha3 structure to its freshly initialized state, continuing with the established heap hint and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha3_512_Free for teardown at end of life.

Parameters:

- **sha3** Pointer to the sha3 structure to be reset.

See:

- wc_InitSha3_512
- wc_Sha3_512_Final
- wc_Sha3_512_Free

Return:

- 0 Returned upon successfully resetting the structure.
- BAD_FUNC_ARG Returned if sha3 is NULL.
- negative Other values may be returned on reinitialization failure in fallback (non-plain-software) configurations.

Example

```
wc_Sha3 sha3;
byte data[] = { Data to be discarded };
byte data2[] = { Data to be hashed };
byte hash[WC_SHA3_512_DIGEST_SIZE];

if ((ret = wc_InitSha3_512(&sha3, NULL, INVALID_DEVID)) == 0) {
    wc_Sha3_512_Update(&sha3, data, sizeof(data));
    wc_Sha3_512_Reset(&sha3); /* discard the partially hashed message */
    wc_Sha3_512_Update(&sha3, data2, sizeof(data2));
    wc_Sha3_512_Final(&sha3, hash);
    wc_Sha3_512_Free(&sha3);
}
```

```
int wc_Sha3_512_GetHash(
    wc_Sha3 * sha3,
    byte * hash
)
```

Gets hash data. Result is placed into hash. Does not reset state of sha3 struct.

Parameters:

- **sha3** pointer to the sha3 structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Sha3_512Hash
- wc_Sha3_512_Final
- wc_InitSha3_512
- wc_Sha3_512_Copy

Return: 0 Returned upon successful copying of the hash.

Example

```
wc_Sha3 sha3[1];
if ((ret = wc_InitSha3_512(sha3, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitSha3_512 failed");
}
else {
   wc_Sha3_512_Update(sha3, data, len);
   wc_Sha3_512_GetHash(sha3, hash);
}
```

```
int wc_Sha3_512_Copy(
    wc_Sha3 * sha3,
    wc_Sha3 * dst
)
```

Copy the state of the hash.

Parameters:

- **sha3** pointer to the sha3 structure to copy
- **dst** pointer to the sha3 structure to copy into (must be zeroed or previously initialized)

See:

- wc_Sha3_512Hash
- wc_Sha3_512_Final
- wc_InitSha3_512
- wc_Sha3_512_GetHash

Return: 0 Returned upon successful copying.

Example

```
wc_Sha3 sha3[1];
wc_Sha3 sha3_dup[1];
memset(sha3_dup, 0, sizeof(*sha3_dup));
if ((ret = wc_InitSha3_512(sha3, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitSha3_512 failed");
}
else {
```

```
    wc_Sha3_512_Update(sha3, data, len);
    wc_Sha3_512_Copy(sha3, sha3_dup);
}



int wc_InitShake128(
    wc_Shake * shake,
    void * heap,
    int devId
)
```

This function initializes SHAKE-128. This is automatically called by wc_Shake128Hash.

Parameters:

- **shake** pointer to the shake structure to use for encryption

See:

- wc_Shake128Hash
- wc_Shake128_Update
- wc_Shake128_Final

Return: 0 Returned upon successfully initializing

Example

```
wc_Shake shake[1];
if ((ret = wc_InitShake128(shake, NULL, INVALID_DEVID)) != 0) {
    WOLFSSL_MSG("wc_InitShake128 failed");
}
else {
    wc_Shake128_Update(shake, data, len);
    wc_Shake128_Final(shake, hash);
}



int wc_Shake128_Update(
    wc_Shake * sha,
    const byte * data,
    word32 len
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **shake** pointer to the shake structure to use for encryption
- **data** the data to be hashed
- **len** length of data to be hashed

See:

- wc_Shake128Hash
- wc_Shake128_Final
- wc_InitShake128

Return: 0 Returned upon successfully adding the data to the digest.

Example

```
wc_Shake shake[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitShake128(shake, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitShake128 failed");
}
else {
    wc_Shake128_Update(shake, data, len);
    wc_Shake128_Final(shake, hash);
}
```

```
int wc_Shake128_Final(
    wc_Shake * shake,
    byte * hash,
    word32 hashLen
)
```

Finalizes hashing of data. Result is placed into hash. Resets state of shake struct.

Parameters:

- **shake** pointer to the shake structure to use for encryption
- **hash** Byte array to hold hash value.
- **hashLen** Number of bytes to write to hash.

See:

- wc_Shake128Hash
- wc_Shake128_GetHash
- wc_InitShake128

Return: 0 Returned upon successfully finalizing.

Example

```
wc_Shake shake[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);
```

```
if ((ret = wc_InitShake128(shake, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitShake128 failed");
}
else {
   wc_Shake128_Update(shake, data, len);
   wc_Shake128_Final(shake, hash);
}
```

```
int wc_Shake128_Absorb(
    wc_Shake * sha,
    const byte * data,
    word32 len
)
```

Called to absorb the provided byte array of length len. Cannot be called incrementally.

Parameters:

- **shake** pointer to the shake structure to use for encryption
- **data** the data to be absorbed
- **len** length of data to be absorbed

See:

- wc_Shake128_SqueezeBlocks
- wc_InitShake128

Return: 0 Returned upon successfully absorbed the data.

Example

```
wc_Shake shake[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);
byte out[2 * WC_SHA3_128_BLOCK_SIZE];
int blocks = 2;

if ((ret = wc_InitShake128(shake, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitShake128 failed");
}
else {
   wc_Shake128_Absorb(shake, data, len);
   wc_Shake128_SqueezeBlocks(shake, out, blocks);
}
```

```
int wc_Shake128_SqueezeBlocks(
    wc_Shake * shake,
    byte * out,
    word32 blockCnt
)
```

Squeeze out more blocks of data. Result is placed into out. Can be called inrementally.

Parameters:

- **shake** pointer to the shake structure to use for encryption
- **hash** Byte array to hold output.
- **blocks** Number of blocks to squeeze out. Each block is WC_SHA3_128_BLOCK_SIZE bytes in length.

See:

- wc_Shake128_Absorb
- wc_InitShake128

Return: 0 Returned upon successfully squeezing.

Example

```
wc_Shake shake[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);
byte out[2 * WC_SHA3_128_BLOCK_SIZE];
int blocks = 2;

if ((ret = wc_InitShake128(shake, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitShake128 failed");
}
else {
   wc_Shake128_Absorb(shake, data, len);
   wc_Shake128_SqueezeBlocks(shake, out, blocks);
}
```

```
void wc_Shake128_Free(
    wc_Shake * shake
)
```

Resets the wc_Shake structure. Note: this is only supported if you have WOLFSSL_TI_HASH defined.

Parameters:

- **shake** Pointer to the shake structure to be freed.

See:

- wc_InitShake128
- wc_Shake128_Update
- wc_Shake128_Final

Return: none No returns.

Example

```
wc_Shake shake;
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitShake128(&shake, NULL, INVALID_DEVID)) != 0) {
    WOLFSSL_MSG("wc_InitShake128 failed");
}
else {
    wc_Shake128_Update(&shake, data, len);
    wc_Shake128_Final(&shake, hash);
    wc_Shake128_Free(&shake);
}
```

```
int wc_Shake128_Reset(
    wc_Shake * shake
)
```

Resets the shake structure to its freshly initialized state, continuing with the established heap hint and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially absorbed input, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Shake128_Free for teardown at end of life.

Parameters:

- **shake** Pointer to the shake structure to be reset.

See:

- wc_InitShake128
- wc_Shake128_Final
- wc_Shake128_Free

Return:

- 0 Returned upon successfully resetting the structure.
- BAD_FUNC_ARG Returned if shake is NULL.
- negative Other values may be returned on reinitialization failure in fallback (non-plain-software) configurations.

Example

```
wc_Shake shake;
byte data[] = { Data to be discarded };
byte data2[] = { Data to be hashed };
byte hash[64];

if ((ret = wc_InitShake128(&shake, NULL, INVALID_DEVID)) == 0) {
    wc_Shake128_Update(&shake, data, sizeof(data));
    wc_Shake128_Reset(&shake); /* discard the partially absorbed input */
```

```
    wc_Shake128_Update(&shake, data2, sizeof(data2));
    wc_Shake128_Final(&shake, hash, sizeof(hash));
    wc_Shake128_Free(&shake);
}
```

```
int wc_Shake128_GetHash(
    wc_Shake * shake,
    byte * hash
)
```

Gets hash data. Result is placed into hash. Does not reset state of shake struct.

Parameters:

- **shake** pointer to the shake structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Shake128Hash
- wc_Shake128_Final
- wc_InitShake128
- wc_Shake128_Copy

Return: 0 Returned upon successful copying of the hash.

Example

```
wc_Shake shake[1];
if ((ret = wc_InitShake128(shake, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitShake128 failed");
}
else {
   wc_Shake128_Update(shake, data, len);
   wc_Shake128_GetHash(shake, hash);
}
```

```
int wc_Shake128_Copy(
    wc_Shake * src,
    wc_Sha3 * dst
)
```

Copy the state of the hash.

Parameters:

- **shake** pointer to the shake structure to copy
- **dst** pointer to the shake structure to copy into (must be zeroed or previously initialized)

See:

- wc_Shake128Hash
- wc_Shake128_Final
- wc_InitShake128
- wc_Shake128_GetHash

Return: 0 Returned upon successful copying.

Example

```
wc_Shake shake[1];
wc_Shake shake_dup[1];
memset(shake_dup, 0, sizeof(*shake_dup));
if ((ret = wc_InitShake128(shake, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitShake128 failed");
}
else {
   wc_Shake128_Update(shake, data, len);
   wc_Shake128_Copy(shake, shake_dup);
}
```

```
int wc_InitShake256(
    wc_Shake * shake,
    void * heap,
    int devId
)
```

This function initializes SHAKE-256. This is automatically called by wc_Shake256Hash.

Parameters:

- **shake** pointer to the shake structure to use for encryption

See:

- wc_Shake256Hash
- wc_Shake256_Update
- wc_Shake256_Final

Return: 0 Returned upon successfully initializing

Example

```
wc_Shake shake[1];
if ((ret = wc_InitShake256(shake, NULL, INVALID_DEVID)) != 0) {
    WOLFSSL_MSG("wc_InitShake256 failed");
}
else {
    wc_Shake256_Update(shake, data, len);
```

```
    wc_Shake256_Final(shake, hash, sizeof(hash));
}
```

```
int wc_Shake256_Update(
    wc_Shake * sha,
    const byte * data,
    word32 len
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **shake** pointer to the shake structure to use for encryption
- **data** the data to be hashed
- **len** length of data to be hashed

See:

- wc_Shake256Hash
- wc_Shake256_Final
- wc_InitShake256

Return: 0 Returned upon successfully adding the data to the digest.

Example

```
wc_Shake shake[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitShake256(shake, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitShake256 failed");
}
else {
    wc_Shake256_Update(shake, data, len);
    wc_Shake256_Final(shake, hash, sizeof(hash));
}
```

```
int wc_Shake256_Final(
    wc_Shake * shake,
    byte * hash,
    word32 hashLen
)
```

Finalizes hashing of data. Result is placed into hash. Resets state of shake struct.

Parameters:

- **shake** pointer to the shake structure to use for encryption
- **hash** Byte array to hold hash value.
- **hashLen** Size of hash in bytes.

See:

- wc_Shake256Hash
- wc_Shake256_GetHash
- wc_InitShake256

Return: 0 Returned upon successfully finalizing.

Example

```
wc_Shake shake[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitShake256(shake, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitShake256 failed");
}
else {
   wc_Shake256_Update(shake, data, len);
   wc_Shake256_Final(shake, hash, sizeof(hash));
}
```

```
int wc_Shake256_Absorb(
    wc_Shake * sha,
    const byte * data,
    word32 len
)
```

Called to absorb the provided byte array of length len. Cannot be called incrementally.

Parameters:

- **shake** pointer to the shake structure to use for encryption
- **data** the data to be absorbed
- **len** length of data to be absorbed

See:

- wc_Shake256_SqueezeBlocks
- wc_InitShake256

Return: 0 Returned upon successfully absorbed the data.

Example

```
wc_Shake shake[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);
byte out[2 * WC_SHA3_256_BLOCK_SIZE];
int blocks = 2;

if ((ret = wc_InitShake256(shake, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitShake256 failed");
}
else {
   wc_Shake256_Absorb(shake, data, len);
   wc_Shake256_SqueezeBlocks(shake, out, blocks);
}
```

```
int wc_Shake256_SqueezeBlocks(
    wc_Shake * shake,
    byte * out,
    word32 blockCnt
)
```

Squeeze out more blocks of data. Result is placed into out. Can be called incrementally.

Parameters:

- **shake** pointer to the shake structure to use for encryption
- **hash** Byte array to hold output.
- **blocks** Number of blocks to squeeze out. Each block is WC_SHA3_256_BLOCK_SIZE bytes in length.

See:

- wc_Shake256_Absorb
- wc_InitShake256

Return: 0 Returned upon successfully squeezing.

Example

```
wc_Shake shake[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);
byte out[2 * WC_SHA3_256_BLOCK_SIZE];
int blocks = 2;

if ((ret = wc_InitShake256(shake, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitShake256 failed");
}
else {
   wc_Shake256_Absorb(shake, data, len);
   wc_Shake256_SqueezeBlocks(shake, out, blocks);
}
```

```
void wc_Shake256_Free(
    wc_Shake * shake
)
```

Resets the wc_Shake structure. Note: this is only supported if you have WOLFSSL_TI_HASH defined.

Parameters:

- **shake** Pointer to the shake structure to be freed.

See:

- wc_InitShake256
- wc_Shake256_Update
- wc_Shake256_Final

Return: none No returns.

Example

```
wc_Shake shake;
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitShake256(&shake, NULL, INVALID_DEVID)) != 0) {
    WOLFSSL_MSG("wc_InitShake256 failed");
}
else {
    wc_Shake256_Update(&shake, data, len);
    wc_Shake256_Final(&shake, hash, sizeof(hash));
    wc_Shake256_Free(&shake);
}
```

```
int wc_Shake256_Reset(
    wc_Shake * shake
)
```

Resets the shake structure to its freshly initialized state, continuing with the established heap hint and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially absorbed input, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Shake256_Free for teardown at end of life.

Parameters:

- **shake** Pointer to the shake structure to be reset.

See:

- wc_InitShake256
- wc_Shake256_Final
- wc_Shake256_Free

Return:

- 0 Returned upon successfully resetting the structure.
- BAD_FUNC_ARG Returned if shake is NULL.
- negative Other values may be returned on reinitialization failure in fallback (non-plain-software) configurations.

Example

```
wc_Shake shake;
byte data[] = { Data to be discarded };
byte data2[] = { Data to be hashed };
byte hash[64];

if ((ret = wc_InitShake256(&shake, NULL, INVALID_DEVID)) == 0) {
    wc_Shake256_Update(&shake, data, sizeof(data));
    wc_Shake256_Reset(&shake); /* discard the partially absorbed input */
    wc_Shake256_Update(&shake, data2, sizeof(data2));
    wc_Shake256_Final(&shake, hash, sizeof(hash));
    wc_Shake256_Free(&shake);
}
```

```
int wc_Shake256_GetHash(
    wc_Shake * shake,
    byte * hash
)
```

Gets hash data. Result is placed into hash. Does not reset state of shake struct.

Parameters:

- **shake** pointer to the shake structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Shake256Hash
- wc_Shake256_Final
- wc_InitShake256
- wc_Shake256_Copy

Return: 0 Returned upon successful copying of the hash.

Example

```
wc_Shake shake[1];
if ((ret = wc_InitShake256(shake, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitShake256 failed");
}
else {
   wc_Shake256_Update(shake, data, len);
   wc_Shake256_GetHash(shake, hash);
}
```

```
int wc_Shake256_Copy(
    wc_Shake * src,
    wc_Sha3 * dst
)
```

Copy the state of the hash.

Parameters:

- **shake** pointer to the shake structure to copy
- **dst** pointer to the shake structure to copy into (must be zeroed or previously initialized)

See:

- wc_Shake256Hash
- wc_Shake256_Final
- wc_InitShake256
- wc_Shake256_GetHash

Return: 0 Returned upon successful copying.

Example

```
wc_Shake shake[1];
wc_Shake shake_dup[1];
memset(shake_dup, 0, sizeof(*shake_dup));
if ((ret = wc_InitShake256(shake, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitShake256 failed");
}
else {
   wc_Shake256_Update(shake, data, len);
   wc_Shake256_Copy(shake, shake_dup);
}
```

```
int wc_InitKmac128(
    wc_Kmac * kmac,
    const byte * key,
    word32 keyLen,
    const byte * custom,
    word32 customLen,
```

```
    void * heap,
    int devId
)
```

This function initializes a KMAC128 operation with a key and an optional customization string, per NIST SP 800-185. KMAC128 requires SHAKE128 (WOLFSSL_SHAKE128) and is enabled with WOLFSSL_KMAC.

Parameters:

- **kmac** pointer to the wc_Kmac structure to initialize.
- **key** the key bytes.
- **keyLen** length of the key in bytes.
- **custom** customization string, or NULL when customLen is 0.
- **customLen** length of the customization string in bytes.
- **heap** pointer to a heap hint, may be NULL.
- **devId** device identifier, or INVALID_DEVID.

See:

- wc_Kmac128_Update
- wc_Kmac128_Final
- wc_Kmac128_FinalXof
- wc_Kmac128_Free
- wc_Kmac128Hash

Return:

- 0 Returned upon successfully initializing.
- BAD_FUNC_ARG Returned when kmac is NULL, or key/custom is NULL with a non-zero length.

Example

```
wc_Kmac kmac;
byte key[32];
byte data[] = { Data to authenticate };
byte out[32];

if (wc_InitKmac128(&kmac, key, sizeof(key), NULL, 0, NULL,
        INVALID_DEVID) == 0) {
    wc_Kmac128_Update(&kmac, data, sizeof(data));
    wc_Kmac128_Final(&kmac, out, sizeof(out));
    wc_Kmac128_Free(&kmac);
}
```

```
int wc_Kmac128_Update(
    wc_Kmac * kmac,
    const byte * in,
    word32 inLen
)
```

This function absorbs message data into a KMAC128 operation. It may be called any number of times before finalizing.

Parameters:

- **kmac** pointer to the wc_Kmac structure holding state.
- **in** the message data to authenticate.
- **inLen** length of the message data in bytes.

See:

- wc_InitKmac128
- wc_Kmac128_Final
- wc_Kmac128_Free

Return:

- 0 Returned upon successfully adding the data.
- BAD_FUNC_ARG Returned when kmac is NULL, or in is NULL with a non-zero length.

Example

```
wc_Kmac kmac;
byte data[] = { Data to authenticate };

if (wc_InitKmac128(&kmac, key, keyLen, NULL, 0, NULL,
        INVALID_DEVID) == 0) {
    wc_Kmac128_Update(&kmac, data, sizeof(data));
    wc_Kmac128_Final(&kmac, out, sizeof(out));
    wc_Kmac128_Free(&kmac);
}
```

```
int wc_Kmac128_Final(
    wc_Kmac * kmac,
    byte * out,
    word32 outLen
)
```

This function finalizes a KMAC128 operation, producing outLen bytes of output. The output length is bound into the result, so a KMAC with a different length is a different value.

Parameters:

- **kmac** pointer to the wc_Kmac structure holding state.
- **out** buffer to hold the output.
- **outLen** number of output bytes to produce.

See:

- wc_InitKmac128
- wc_Kmac128_Update
- wc_Kmac128_FinalXof
- wc_Kmac128_Free

Return:

- 0 Returned upon successfully finalizing.
- BAD_FUNC_ARG Returned when kmac is NULL, or out is NULL with a non-zero length.

Example

```
byte out[32];
wc_Kmac128_Final(&kmac, out, sizeof(out));
```

```
int wc_Kmac128_FinalXof(
    wc_Kmac * kmac,
    byte * out,
    word32 outLen
)
```

This function finalizes a KMAC128 operation as an XOF (KMACXOF128). Unlike wc_Kmac128_Final, the output length is not bound into the result, so any amount of output may be requested.

Parameters:

- **kmac** pointer to the wc_Kmac structure holding state.
- **out** buffer to hold the output.
- **outLen** number of output bytes to produce.

See:

- wc_InitKmac128
- wc_Kmac128_Update
- wc_Kmac128_Final
- wc_Kmac128_Free

Return:

- 0 Returned upon successfully finalizing.
- BAD_FUNC_ARG Returned when kmac is NULL, or out is NULL with a non-zero length.

Example

```
byte out[64];
wc_Kmac128_FinalXof(&kmac, out, sizeof(out));
```

```
void wc_Kmac128_Free(
    wc_Kmac * kmac
)
```

This function releases any resources associated with a KMAC128 operation. Passing NULL is safe and does nothing.

Parameters:

- **kmac** pointer to the wc_Kmac structure to free, may be NULL.

See: wc_InitKmac128

Example

```
wc_Kmac128_Free(&kmac);
```

```
int wc_Kmac128Hash(
    const byte * key,
    word32 keyLen,
    const byte * custom,
    word32 customLen,
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen
)
```

This function computes a KMAC128 over a single message in one call, initializing, updating and finalizing internally.

Parameters:

- **key** the key bytes.
- **keyLen** length of the key in bytes.
- **custom** customization string, or NULL when customLen is 0.
- **customLen** length of the customization string in bytes.
- **in** the message data to authenticate.
- **inLen** length of the message data in bytes.
- **out** buffer to hold the output.
- **outLen** number of output bytes to produce.

See:

- wc_InitKmac128
- wc_Kmac128_Update
- wc_Kmac128_Final

Return:

- 0 Returned upon success.
- BAD_FUNC_ARG Returned when a required pointer is NULL with a non-zero length.

Example

```
byte key[32];
byte data[] = { Data to authenticate };
byte out[32];

wc_Kmac128Hash(key, sizeof(key), NULL, 0, data, sizeof(data),
    out, sizeof(out));
```

```
int wc_Kmac128HashXof(
    const byte * key,
    word32 keyLen,
    const byte * custom,
    word32 customLen,
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen
)
```

This function computes a KMACXOF128 over a single message in one call. Like wc_Kmac128Hash but the output length is not bound into the result, so any amount of output may be requested.

Parameters:

- **key** the key bytes.
- **keyLen** length of the key in bytes.
- **custom** customization string, or NULL when customLen is 0.
- **customLen** length of the customization string in bytes.
- **in** the message data to authenticate.
- **inLen** length of the message data in bytes.
- **out** buffer to hold the output.
- **outLen** number of output bytes to produce.

See:

- wc_InitKmac128
- wc_Kmac128_FinalXof
- wc_Kmac128Hash

Return:

- 0 Returned upon success.
- BAD_FUNC_ARG Returned when a required pointer is NULL with a non-zero length.

Example

```
byte key[32];
byte data[] = { Data to authenticate };
byte out[64];

wc_Kmac128HashXof(key, sizeof(key), NULL, 0, data, sizeof(data),
    out, sizeof(out));
```

```
int wc_InitKmac256(
    wc_Kmac * kmac,
    const byte * key,
    word32 keyLen,
    const byte * custom,
    word32 customLen,
    void * heap,
    int devId
)
```

This function initializes a KMAC256 operation with a key and an optional customization string, per NIST SP 800-185. KMAC256 requires SHAKE256 (WOLFSSL_SHAKE256) and is enabled with WOLFSSL_KMAC.

Parameters:

- **kmac** pointer to the wc_Kmac structure to initialize.
- **key** the key bytes.
- **keyLen** length of the key in bytes.
- **custom** customization string, or NULL when customLen is 0.
- **customLen** length of the customization string in bytes.
- **heap** pointer to a heap hint, may be NULL.
- **devId** device identifier, or INVALID_DEVID.

See:

- wc_Kmac256_Update
- wc_Kmac256_Final
- wc_Kmac256_FinalXof
- wc_Kmac256_Free
- wc_Kmac256Hash

Return:

- 0 Returned upon successfully initializing.
- BAD_FUNC_ARG Returned when kmac is NULL, or key/custom is NULL with a non-zero length.

Example

```
wc_Kmac kmac;
byte key[32];
byte data[] = { Data to authenticate };
byte out[64];
```

```
if (wc_InitKmac256(&kmac, key, sizeof(key), NULL, 0, NULL,
        INVALID_DEVID) == 0) {
    wc_Kmac256_Update(&kmac, data, sizeof(data));
    wc_Kmac256_Final(&kmac, out, sizeof(out));
    wc_Kmac256_Free(&kmac);
}
```

```
int wc_Kmac256_Update(
    wc_Kmac * kmac,
    const byte * in,
    word32 inLen
)
```

This function absorbs message data into a KMAC256 operation. It may be called any number of times before finalizing.

Parameters:

- **kmac** pointer to the wc_Kmac structure holding state.
- **in** the message data to authenticate.
- **inLen** length of the message data in bytes.

See:

- wc_InitKmac256
- wc_Kmac256_Final
- wc_Kmac256_Free

Return:

- 0 Returned upon successfully adding the data.
- BAD_FUNC_ARG Returned when kmac is NULL, or in is NULL with a non-zero length.

Example

```
wc_Kmac256_Update(&kmac, data, sizeof(data));
```

```
int wc_Kmac256_Final(
    wc_Kmac * kmac,
    byte * out,
    word32 outLen
)
```

This function finalizes a KMAC256 operation, producing outLen bytes of output. The output length is bound into the result, so a KMAC with a different length is a different value.

Parameters:

- **kmac** pointer to the wc_Kmac structure holding state.
- **out** buffer to hold the output.
- **outLen** number of output bytes to produce.

See:

- wc_InitKmac256
- wc_Kmac256_Update
- wc_Kmac256_FinalXof
- wc_Kmac256_Free

Return:

- 0 Returned upon successfully finalizing.
- BAD_FUNC_ARG Returned when kmac is NULL, or out is NULL with a non-zero length.

Example

```
byte out[64];
wc_Kmac256_Final(&kmac, out, sizeof(out));
```

```
int wc_Kmac256_FinalXof(
    wc_Kmac * kmac,
    byte * out,
    word32 outLen
)
```

This function finalizes a KMAC256 operation as an XOF (KMACXOF256). Unlike wc_Kmac256_Final, the output length is not bound into the result, so any amount of output may be requested.

Parameters:

- **kmac** pointer to the wc_Kmac structure holding state.
- **out** buffer to hold the output.
- **outLen** number of output bytes to produce.

See:

- wc_InitKmac256
- wc_Kmac256_Update
- wc_Kmac256_Final
- wc_Kmac256_Free

Return:

- 0 Returned upon successfully finalizing.
- BAD_FUNC_ARG Returned when kmac is NULL, or out is NULL with a non-zero length.

Example

```
byte out[128];
wc_Kmac256_FinalXof(&kmac, out, sizeof(out));
```

```
void wc_Kmac256_Free(
    wc_Kmac * kmac
)
```

This function releases any resources associated with a KMAC256 operation. Passing NULL is safe and does nothing.

Parameters:

- **kmac** pointer to the wc_Kmac structure to free, may be NULL.

See: wc_InitKmac256

Example

```
wc_Kmac256_Free(&kmac);
```

```
int wc_Kmac256Hash(
    const byte * key,
    word32 keyLen,
    const byte * custom,
    word32 customLen,
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen
)
```

This function computes a KMAC256 over a single message in one call, initializing, updating and finalizing internally.

Parameters:

- **key** the key bytes.
- **keyLen** length of the key in bytes.
- **custom** customization string, or NULL when customLen is 0.
- **customLen** length of the customization string in bytes.
- **in** the message data to authenticate.
- **inLen** length of the message data in bytes.
- **out** buffer to hold the output.
- **outLen** number of output bytes to produce.

See:

- wc_InitKmac256
- wc_Kmac256_Update
- wc_Kmac256_Final

Return:

- 0 Returned upon success.
- BAD_FUNC_ARG Returned when a required pointer is NULL with a non-zero length.

Example

```
byte key[32];
byte data[] = { Data to authenticate };
byte out[64];

wc_Kmac256Hash(key, sizeof(key), NULL, 0, data, sizeof(data),
    out, sizeof(out));
```

```
int wc_Kmac256HashXof(
    const byte * key,
    word32 keyLen,
    const byte * custom,
    word32 customLen,
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen
)
```

This function computes a KMACXOF256 over a single message in one call. Like wc_Kmac256Hash but the output length is not bound into the result, so any amount of output may be requested.

Parameters:

- **key** the key bytes.
- **keyLen** length of the key in bytes.
- **custom** customization string, or NULL when customLen is 0.
- **customLen** length of the customization string in bytes.
- **in** the message data to authenticate.
- **inLen** length of the message data in bytes.
- **out** buffer to hold the output.
- **outLen** number of output bytes to produce.

See:

- wc_InitKmac256
- wc_Kmac256_FinalXof
- wc_Kmac256Hash

Return:

- 0 Returned upon success.
- BAD_FUNC_ARG Returned when a required pointer is NULL with a non-zero length.

Example

```
byte key[32];
byte data[] = { Data to authenticate };
byte out[128];

wc_Kmac256HashXof(key, sizeof(key), NULL, 0, data, sizeof(data),
    out, sizeof(out));
```

```
int wc_Kmac128_Copy(
    wc_Kmac * src,
    wc_Kmac * dst
)
```

This function copies the state of a KMAC128 operation so it can be finalized more than once, for example over a common message prefix.

Parameters:

- **src** pointer to the wc_Kmac structure to copy from.
- **dst** pointer to the wc_Kmac structure to copy into.

See:

- wc_InitKmac128
- wc_Kmac128_Update
- wc_Kmac128_Final

Return:

- 0 Returned upon success.
- BAD_FUNC_ARG Returned when src or dst is NULL.

Example

```
wc_Kmac kmac;
wc_Kmac copy;

wc_InitKmac128(&kmac, key, keyLen, NULL, 0, NULL, INVALID_DEVID);
wc_Kmac128_Update(&kmac, prefix, prefixLen);
wc_Kmac128_Copy(&kmac, &copy);
```

```
int wc_Kmac256_Copy(
    wc_Kmac * src,
    wc_Kmac * dst
)
```

This function copies the state of a KMAC256 operation so it can be finalized more than once, for example over a common message prefix.

Parameters:

- **src** pointer to the wc_Kmac structure to copy from.
- **dst** pointer to the wc_Kmac structure to copy into.

See:

- wc_InitKmac256
- wc_Kmac256_Update
- wc_Kmac256_Final

Return:

- 0 Returned upon success.
- BAD_FUNC_ARG Returned when src or dst is NULL.

Example

```
wc_Kmac kmac;
wc_Kmac copy;

wc_InitKmac256(&kmac, key, keyLen, NULL, 0, NULL, INVALID_DEVID);
wc_Kmac256_Update(&kmac, prefix, prefixLen);
wc_Kmac256_Copy(&kmac, &copy);
```

```
int wc_InitCshake128(
    wc_Cshake * cshake,
    const byte * name,
    word32 nameLen,
    const byte * custom,
    word32 customLen,
    void * heap,
    int devId
)
```

This function initializes a cSHAKE128 operation (customizable SHAKE, NIST SP 800-185) with a function-name and customization string. cSHAKE is enabled together with KMAC (WOLFSSL_KMAC). When both strings are empty, cSHAKE reduces to plain SHAKE128.

Parameters:

- **cshake** pointer to the wc_Cshake structure to initialize.

- **name** function-name string (reserved for NIST-defined functions); use an empty string for application customization via custom.
- **nameLen** length of the function-name string in bytes.
- **custom** customization string, or NULL when customLen is 0.
- **customLen** length of the customization string in bytes.
- **heap** pointer to a heap hint, may be NULL.
- **devId** device identifier, or INVALID_DEVID.

See:

- wc_Cshake128_Update
- wc_Cshake128_Final
- wc_Cshake128_Free
- wc_Cshake128

Return:

- 0 Returned upon successfully initializing.
- BAD_FUNC_ARG Returned when a required pointer is NULL.

Example

```
wc_Cshake cshake;
byte out[32];

if (wc_InitCshake128(&cshake, NULL, 0, custom, customLen, NULL,
        INVALID_DEVID) == 0) {
    wc_Cshake128_Update(&cshake, data, dataLen);
    wc_Cshake128_Final(&cshake, out, sizeof(out));
    wc_Cshake128_Free(&cshake);
}
```

```
int wc_Cshake128_Update(
    wc_Cshake * cshake,
    const byte * in,
    word32 inLen
)
```

This function absorbs message data into a cSHAKE128 operation.

Parameters:

- **cshake** pointer to the wc_Cshake structure holding state.
- **in** the message data.
- **inLen** length of the message data in bytes.

See:

- wc_InitCshake128
- wc_Cshake128_Final

Return:

- 0 Returned upon successfully adding the data.
- BAD_FUNC_ARG Returned when cshake is NULL, or in is NULL with a non-zero length.

Example

```
wc_Cshake128_Update(&cshake, data, dataLen);
```

```
int wc_Cshake128_Final(
    wc_Cshake * cshake,
    byte * out,
    word32 outLen
)
```

This function finalizes a cSHAKE128 operation, squeezing outLen bytes of output. cSHAKE is an XOF, so a longer squeeze extends a shorter one over the same input.

Parameters:

- **cshake** pointer to the wc_Cshake structure holding state.
- **out** buffer to hold the output.
- **outLen** number of output bytes to produce.

See:

- wc_InitCshake128
- wc_Cshake128_Update

Return:

- 0 Returned upon successfully finalizing.
- BAD_FUNC_ARG Returned when cshake is NULL, or out is NULL with a non-zero length.

Example

```
byte out[32];
wc_Cshake128_Final(&cshake, out, sizeof(out));
```

```
void wc_Cshake128_Free(
    wc_Cshake * cshake
)
```

This function releases any resources associated with a cSHAKE128 operation. Passing NULL is safe and does nothing.

Parameters:

- **cshake** pointer to the wc_Cshake structure to free, may be NULL.

See: wc_InitCshake128

Example

```
wc_Cshake128_Free(&cshake);
```

```
int wc_Cshake128(
    const byte * name,
    word32 nameLen,
    const byte * custom,
    word32 customLen,
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen
)
```

This function computes a cSHAKE128 over a single message in one call, initializing, updating and finalizing internally.

Parameters:

- **name** function-name string, or NULL when nameLen is 0.
- **nameLen** length of the function-name string in bytes.
- **custom** customization string, or NULL when customLen is 0.
- **customLen** length of the customization string in bytes.
- **in** the message data.
- **inLen** length of the message data in bytes.
- **out** buffer to hold the output.
- **outLen** number of output bytes to produce.

See:

- wc_InitCshake128
- wc_Cshake128_Final

Return:

- 0 Returned upon success.
- BAD_FUNC_ARG Returned when a required pointer is NULL with a non-zero length.

Example

```
byte out[32];
wc_Cshake128(NULL, 0, custom, customLen, data, dataLen, out, sizeof(out));
```

```
int wc_InitCshake256(
    wc_Cshake * cshake,
    const byte * name,
    word32 nameLen,
    const byte * custom,
    word32 customLen,
    void * heap,
    int devId
)
```

This function initializes a cSHAKE256 operation (customizable SHAKE, NIST SP 800-185) with a function-name and customization string. cSHAKE is enabled together with KMAC (WOLFSSL_KMAC). When both strings are empty, cSHAKE reduces to plain SHAKE256.

Parameters:

- **cshake** pointer to the wc_Cshake structure to initialize.
- **name** function-name string (reserved for NIST-defined functions); use an empty string for application customization via custom.
- **nameLen** length of the function-name string in bytes.
- **custom** customization string, or NULL when customLen is 0.
- **customLen** length of the customization string in bytes.
- **heap** pointer to a heap hint, may be NULL.
- **devId** device identifier, or INVALID_DEVID.

See:

- wc_Cshake256_Update
- wc_Cshake256_Final
- wc_Cshake256_Free
- wc_Cshake256

Return:

- 0 Returned upon successfully initializing.
- BAD_FUNC_ARG Returned when a required pointer is NULL.

Example

```
wc_Cshake cshake;
byte out[32];

if (wc_InitCshake256(&cshake, NULL, 0, custom, customLen, NULL,
        INVALID_DEVID) == 0) {
    wc_Cshake256_Update(&cshake, data, dataLen);
    wc_Cshake256_Final(&cshake, out, sizeof(out));
    wc_Cshake256_Free(&cshake);
}
```

```
int wc_Cshake256_Update(
    wc_Cshake * cshake,
    const byte * in,
    word32 inLen
)
```

This function absorbs message data into a cSHAKE256 operation.

Parameters:

- **cshake** pointer to the wc_Cshake structure holding state.
- **in** the message data.
- **inLen** length of the message data in bytes.

See:

- wc_InitCshake256
- wc_Cshake256_Final

Return:

- 0 Returned upon successfully adding the data.
- BAD_FUNC_ARG Returned when cshake is NULL, or in is NULL with a non-zero length.

Example

```
wc_Cshake256_Update(&cshake, data, dataLen);
```

```
int wc_Cshake256_Final(
    wc_Cshake * cshake,
    byte * out,
    word32 outLen
)
```

This function finalizes a cSHAKE256 operation, squeezing outLen bytes of output. cSHAKE is an XOF, so a longer squeeze extends a shorter one over the same input.

Parameters:

- **cshake** pointer to the wc_Cshake structure holding state.
- **out** buffer to hold the output.
- **outLen** number of output bytes to produce.

See:

- wc_InitCshake256
- wc_Cshake256_Update

Return:

- 0 Returned upon successfully finalizing.
- BAD_FUNC_ARG Returned when cshake is NULL, or out is NULL with a non-zero length.

Example

```
byte out[32];
wc_Cshake256_Final(&cshake, out, sizeof(out));
```

```
void wc_Cshake256_Free(
    wc_Cshake * cshake
)
```

This function releases any resources associated with a cSHAKE256 operation. Passing NULL is safe and does nothing.

Parameters:

- **cshake** pointer to the wc_Cshake structure to free, may be NULL.

See: wc_InitCshake256

Example

```
wc_Cshake256_Free(&cshake);
```

```
int wc_Cshake256(
    const byte * name,
    word32 nameLen,
    const byte * custom,
    word32 customLen,
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen
)
```

This function computes a cSHAKE256 over a single message in one call, initializing, updating and finalizing internally.

Parameters:

- **name** function-name string, or NULL when nameLen is 0.
- **nameLen** length of the function-name string in bytes.
- **custom** customization string, or NULL when customLen is 0.
- **customLen** length of the customization string in bytes.
- **in** the message data.

- **inLen** length of the message data in bytes.
- **out** buffer to hold the output.
- **outLen** number of output bytes to produce.

See:

- wc_InitCshake256
- wc_Cshake256_Final

Return:

- 0 Returned upon success.
- BAD_FUNC_ARG Returned when a required pointer is NULL with a non-zero length.

Example

```
byte out[32];
wc_Cshake256(NULL, 0, custom, customLen, data, dataLen, out, sizeof(out));
```

```
int wc_Cshake128_Copy(
    wc_Cshake * src,
    wc_Cshake * dst
)
```

This function copies the state of a cSHAKE128 operation so it can be finalized more than once, for example over a common message prefix. dst must already be an initialized wc_Cshake.

Parameters:

- **src** pointer to the wc_Cshake structure to copy from.
- **dst** pointer to the wc_Cshake structure to copy into.

See:

- wc_InitCshake128
- wc_Cshake128_Update
- wc_Cshake128_Final

Return:

- 0 Returned upon success.
- BAD_FUNC_ARG Returned when src or dst is NULL.

Example

```
wc_Cshake cshake;
wc_Cshake copy;

wc_InitCshake128(&cshake, NULL, 0, custom, customLen, NULL, INVALID_DEVID);
```

```
wc_InitCshake128(&copy, NULL, 0, custom, customLen, NULL, INVALID_DEVID);
wc_Cshake128_Update(&cshake, prefix, prefixLen);
wc_Cshake128_Copy(&cshake, &copy);
```

```
int wc_Cshake256_Copy(
    wc_Cshake * src,
    wc_Cshake * dst
)
```

This function copies the state of a cSHAKE256 operation so it can be finalized more than once, for example over a common message prefix. dst must already be an initialized wc_Cshake.

Parameters:

- **src** pointer to the wc_Cshake structure to copy from.
- **dst** pointer to the wc_Cshake structure to copy into.

See:

- wc_InitCshake256
- wc_Cshake256_Update
- wc_Cshake256_Final

Return:

- 0 Returned upon success.
- BAD_FUNC_ARG Returned when src or dst is NULL.

Example

```
wc_Cshake cshake;
wc_Cshake copy;

wc_InitCshake256(&cshake, NULL, 0, custom, customLen, NULL, INVALID_DEVID);
wc_InitCshake256(&copy, NULL, 0, custom, customLen, NULL, INVALID_DEVID);
wc_Cshake256_Update(&cshake, prefix, prefixLen);
wc_Cshake256_Copy(&cshake, &copy);
```

```
int wc_InitSha512(
    wc_Sha512 * sha
)
```

This function initializes SHA512. This is automatically called by wc_Sha512Hash.

Parameters:

- **sha512** pointer to the sha512 structure to use for encryption

See:

- wc_Sha512Hash
- wc_Sha512Update
- wc_Sha512Final

Return: 0 Returned upon successfully initializing

Example

```
Sha512 sha512[1];
if ((ret = wc_InitSha512(sha512)) != 0) {
   WOLFSSL_MSG("wc_InitSha512 failed");
}
else {
   wc_Sha512Update(sha512, data, len);
   wc_Sha512Final(sha512, hash);
}
```

```
int wc_Sha512Update(
    wc_Sha512 * sha,
    const byte * data,
    word32 len
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **sha512** pointer to the sha512 structure to use for encryption
- **data** the data to be hashed
- **len** length of data to be hashed

See:

- wc_Sha512Hash
- wc_Sha512Final
- wc_InitSha512

Return: 0 Returned upon successfully adding the data to the digest.

Example

```
Sha512 sha512[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha512(sha512)) != 0) {
   WOLFSSL_MSG("wc_InitSha512 failed");
}
```

```
else {
   wc_Sha512Update(sha512, data, len);
   wc_Sha512Final(sha512, hash);
}
```

```
int wc_Sha512Final(
    wc_Sha512 * sha512,
    byte * hash
)
```

Finalizes hashing of data. Result is placed into hash.

Parameters:

- **sha512** pointer to the sha512 structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Sha512Hash
- wc_Sha512Final
- wc_InitSha512

Return: 0 Returned upon successfully finalizing the hash.

Example

```
Sha512 sha512[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha512(sha512)) != 0) {
    WOLFSSL_MSG("wc_InitSha512 failed");
}
else {
   wc_Sha512Update(sha512, data, len);
   wc_Sha512Final(sha512, hash);
}
```

```
int wc_InitSha384(
    wc_Sha384 * sha
)
```

This function initializes SHA384. This is automatically called by wc_Sha384Hash.

Parameters:

- **sha384** pointer to the sha384 structure to use for encryption

See:

- wc_Sha384Hash
- wc_Sha384Update
- wc_Sha384Final

Return: 0 Returned upon successfully initializing

Example

```
Sha384 sha384[1];
if ((ret = wc_InitSha384(sha384)) != 0) {
   WOLFSSL_MSG("wc_InitSha384 failed");
}
else {
   wc_Sha384Update(sha384, data, len);
   wc_Sha384Final(sha384, hash);
}
```

```
int wc_Sha384Update(
    wc_Sha384 * sha,
    const byte * data,
    word32 len
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **sha384** pointer to the sha384 structure to use for encryption
- **data** the data to be hashed
- **len** length of data to be hashed

See:

- wc_Sha384Hash
- wc_Sha384Final
- wc_InitSha384

Return: 0 Returned upon successfully adding the data to the digest.

Example

```
Sha384 sha384[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha384(sha384)) != 0) {
   WOLFSSL_MSG("wc_InitSha384 failed");
}
```

```
else {
   wc_Sha384Update(sha384, data, len);
   wc_Sha384Final(sha384, hash);
}
```

```
int wc_Sha384Final(
    wc_Sha384 * sha384,
    byte * hash
)
```

Finalizes hashing of data. Result is placed into hash.

Parameters:

- **sha384** pointer to the sha384 structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Sha384Hash
- wc_Sha384Final
- wc_InitSha384

Return: 0 Returned upon successfully finalizing.

Example

```
Sha384 sha384[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha384(sha384)) != 0) {
   WOLFSSL_MSG("wc_InitSha384 failed");
}
else {
   wc_Sha384Update(sha384, data, len);
   wc_Sha384Final(sha384, hash);
}
```

```
int wc_InitSha512_ex(
    wc_Sha512 * sha,
    void * heap,
    int devId
)
```

Initializes SHA512 with heap and device ID.

Parameters:

- **sha** SHA512 structure
- **heap** Heap hint
- **devId** Device ID

See: wc_InitSha512

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
int ret = wc_InitSha512_ex(&sha, NULL, INVALID_DEVID);
```

```
int wc_Sha512FinalRaw(
    wc_Sha512 * sha512,
    byte * hash
)
```

Gets raw hash without finalizing.

Parameters:

- **sha512** SHA512 structure
- **hash** Output hash buffer

See: wc_Sha512Final

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
byte hash[WC_SHA512_DIGEST_SIZE];
int ret = wc_Sha512FinalRaw(&sha, hash);
```

```
void wc_Sha512Free(
    wc_Sha512 * sha
)
```

Frees SHA512 resources.

Parameters:

- **sha** SHA512 structure

See: wc_InitSha512

Return: none No returns

Example

```
wc_Sha512 sha;
wc_InitSha512(&sha);
wc_Sha512Free(&sha);
```

```
int wc_Sha512Reset(
    wc_Sha512 * sha
)
```

Resets the sha structure to its freshly initialized state, continuing with its established allocations, heap hint, and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha512Free for teardown at end of life.

Parameters:

- **sha** Pointer to the sha structure to be reset.

See:

- wc_InitSha512
- wc_Sha512Final
- wc_Sha512Free

Return:

- 0 Returned upon successfully resetting the structure.
- BAD_FUNC_ARG Returned if sha is NULL.
- negative Other values may be returned on reinitialization failure in fallback (non-plain-software) configurations.

Example

```
wc_Sha512 sha;
byte data[] = { Data to be discarded };
byte data2[] = { Data to be hashed };
byte hash[WC_SHA512_DIGEST_SIZE];

if ((ret = wc_InitSha512(&sha)) == 0) {
    wc_Sha512Update(&sha, data, sizeof(data));
    wc_Sha512Reset(&sha); /* discard the partially hashed message */
    wc_Sha512Update(&sha, data2, sizeof(data2));
    wc_Sha512Final(&sha, hash);
```

```
    wc_Sha512Free(&sha);
}



int wc_Sha512GetHash(
    wc_Sha512 * sha512,
    byte * hash
)
```

Gets SHA512 hash without finalizing.

Parameters:

- **sha512** SHA512 structure
- **hash** Output hash buffer

See: wc_Sha512Final

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
byte hash[WC_SHA512_DIGEST_SIZE];
int ret = wc_Sha512GetHash(&sha, hash);



int wc_Sha512Copy(
    wc_Sha512 * src,
    wc_Sha512 * dst
)
```

Copies SHA512 context.

Parameters:

- **src** Source SHA512 structure
- **dst** Destination SHA512 structure (must be zeroed or previously initialized)

See: wc_InitSha512

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 src, dst;
memset(&dst, 0, sizeof(dst));
int ret = wc_InitSha512(&src);
if (ret == 0)
    ret = wc_Sha512Copy(&src, &dst);
```

```
int wc_Sha512_Grow(
    wc_Sha512 * sha512,
    const byte * in,
    int inSz
)
```

Grows SHA512 buffer with input data.

Parameters:

- **sha512** SHA512 structure
- **in** Input data
- **inSz** Input size

See: wc_Sha512Update

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
byte data[100];
int ret = wc_Sha512_Grow(&sha, data, sizeof(data));
```

```
int wc_Sha512SetFlags(
    wc_Sha512 * sha512,
    word32 flags
)
```

Sets SHA512 flags.

Parameters:

- **sha512** SHA512 structure
- **flags** Flags to set

See: wc_InitSha512

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
int ret = wc_Sha512SetFlags(&sha, WC_HASH_FLAG_WILLCOPY);
```

```
int wc_Sha512GetFlags(
    wc_Sha512 * sha512,
    word32 * flags
)
```

Gets SHA512 flags.

Parameters:

- **sha512** SHA512 structure
- **flags** Pointer to store flags

See: wc_Sha512SetFlags

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
word32 flags;
int ret = wc_Sha512GetFlags(&sha, &flags);
```

```
int wc_Sha512Transform(
    wc_Sha512 * sha,
    const unsigned char * data
)
```

Transforms SHA512 block.

Parameters:

- **sha** SHA512 structure
- **data** Block data

See: wc_Sha512Update

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
unsigned char block[WC_SHA512_BLOCK_SIZE];
int ret = wc_Sha512Transform(&sha, block);
```

```
int wc_InitSha512_224(
    wc_Sha512 * sha
)
```

Initializes SHA512/224.

Parameters:

- **sha** SHA512 structure

See: wc_Sha512_224Update

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
int ret = wc_InitSha512_224(&sha);
```

```
int wc_InitSha512_224_ex(
    wc_Sha512 * sha,
    void * heap,
    int devId
)
```

Initializes SHA512/224 with heap and device ID.

Parameters:

- **sha** SHA512 structure
- **heap** Heap hint
- **devId** Device ID

See: wc_InitSha512_224

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
int ret = wc_InitSha512_224_ex(&sha, NULL, INVALID_DEVID);
```

```
int wc_Sha512_224Update(
    wc_Sha512 * sha,
    const byte * data,
    word32 len
)
```

Updates SHA512/224 hash with data.

Parameters:

- **sha** SHA512 structure
- **data** Input data
- **len** Input size

See: wc_InitSha512_224

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
byte data[100];
int ret = wc_Sha512_224Update(&sha, data, sizeof(data));
```

```
int wc_Sha512_224FinalRaw(
    wc_Sha512 * sha512,
    byte * hash
)
```

Gets raw SHA512/224 hash without finalizing.

Parameters:

- **sha512** SHA512 structure
- **hash** Output hash buffer

See: wc_Sha512_224Final

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
byte hash[WC_SHA512_224_DIGEST_SIZE];
int ret = wc_Sha512_224FinalRaw(&sha, hash);
```

```
int wc_Sha512_224Final(
    wc_Sha512 * sha512,
    byte * hash
)
```

Finalizes SHA512/224 hash.

Parameters:

- **sha512** SHA512 structure
- **hash** Output hash buffer

See: wc_Sha512_224Update

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
byte hash[WC_SHA512_224_DIGEST_SIZE];
int ret = wc_Sha512_224Final(&sha, hash);
```

```
void wc_Sha512_224Free(
    wc_Sha512 * sha
)
```

Frees SHA512/224 resources.

Parameters:

- **sha** SHA512 structure

See: wc_InitSha512_224

Return: none No returns

Example

```
wc_Sha512 sha;
wc_InitSha512_224(&sha);
wc_Sha512_224Free(&sha);
```

```
int wc_Sha512_224Reset(
    wc_Sha512 * sha
)
```

Resets the sha structure to its freshly initialized state, continuing with its established allocations, heap hint, and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha512_224Free for teardown at end of life.

Parameters:

- **sha** Pointer to the sha structure to be reset.

See:

- wc_InitSha512_224
- wc_Sha512_224Final
- wc_Sha512_224Free

Return:

- 0 Returned upon successfully resetting the structure.
- BAD_FUNC_ARG Returned if sha is NULL.
- negative Other values may be returned on reinitialization failure in fallback (non-plain-software) configurations.

Example

```
wc_Sha512 sha;
byte data[] = { Data to be discarded };
byte data2[] = { Data to be hashed };
byte hash[WC_SHA512_224_DIGEST_SIZE];

if ((ret = wc_InitSha512_224(&sha)) == 0) {
    wc_Sha512_224Update(&sha, data, sizeof(data));
    wc_Sha512_224Reset(&sha); /* discard the partially hashed message */
    wc_Sha512_224Update(&sha, data2, sizeof(data2));
    wc_Sha512_224Final(&sha, hash);
    wc_Sha512_224Free(&sha);
}
```

```
int wc_Sha512_224GetHash(
    wc_Sha512 * sha512,
    byte * hash
)
```

Gets SHA512/224 hash without finalizing.

Parameters:

- **sha512** SHA512 structure
- **hash** Output hash buffer

See: wc_Sha512_224Final

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
byte hash[WC_SHA512_224_DIGEST_SIZE];
int ret = wc_Sha512_224GetHash(&sha, hash);
```

```
int wc_Sha512_224Copy(
    wc_Sha512 * src,
    wc_Sha512 * dst
)
```

Copies SHA512/224 context.

Parameters:

- **src** Source SHA512 structure
- **dst** Destination SHA512 structure (must be zeroed or previously initialized)

See: wc_InitSha512_224

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 src, dst;
memset(&dst, 0, sizeof(dst));
int ret = wc_InitSha512_224(&src);
if (ret == 0)
    ret = wc_Sha512_224Copy(&src, &dst);
```

```
int wc_Sha512_224SetFlags(
    wc_Sha512 * sha512,
    word32 flags
)
```

Sets SHA512/224 flags.

Parameters:

- **sha512** SHA512 structure
- **flags** Flags to set

See: wc_InitSha512_224

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
int ret = wc_Sha512_224SetFlags(&sha, WC_HASH_FLAG_WILLCOPY);
```

```
int wc_Sha512_224GetFlags(
    wc_Sha512 * sha512,
    word32 * flags
)
```

Gets SHA512/224 flags.

Parameters:

- **sha512** SHA512 structure
- **flags** Pointer to store flags

See: wc_Sha512_224SetFlags

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
word32 flags;
int ret = wc_Sha512_224GetFlags(&sha, &flags);
```

```
int wc_Sha512_224Transform(
    wc_Sha512 * sha,
    const unsigned char * data
)
```

Transforms SHA512/224 block.

Parameters:

- **sha** SHA512 structure
- **data** Block data

See: wc_Sha512_224Update

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
unsigned char block[WC_SHA512_BLOCK_SIZE];
int ret = wc_Sha512_224Transform(&sha, block);
```

```
int wc_InitSha512_256(
    wc_Sha512 * sha
)
```

Initializes SHA512/256.

Parameters:

- **sha** SHA512 structure

See: wc_Sha512_256Update

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
int ret = wc_InitSha512_256(&sha);
```

```
int wc_InitSha512_256_ex(
    wc_Sha512 * sha,
    void * heap,
    int devId
)
```

Initializes SHA512/256 with heap and device ID.

Parameters:

- **sha** SHA512 structure
- **heap** Heap hint
- **devId** Device ID

See: wc_InitSha512_256

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
int ret = wc_InitSha512_256_ex(&sha, NULL, INVALID_DEVID);
```

```
int wc_Sha512_256Update(
    wc_Sha512 * sha,
    const byte * data,
    word32 len
)
```

Updates SHA512/256 hash with data.

Parameters:

- **sha** SHA512 structure
- **data** Input data
- **len** Input size

See: wc_InitSha512_256

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
byte data[100];
int ret = wc_Sha512_256Update(&sha, data, sizeof(data));
```

```
int wc_Sha512_256FinalRaw(
    wc_Sha512 * sha512,
    byte * hash
)
```

Gets raw SHA512/256 hash without finalizing.

Parameters:

- **sha512** SHA512 structure
- **hash** Output hash buffer

See: wc_Sha512_256Final

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
byte hash[WC_SHA512_256_DIGEST_SIZE];
int ret = wc_Sha512_256FinalRaw(&sha, hash);
```

```
int wc_Sha512_256Final(
    wc_Sha512 * sha512,
    byte * hash
)
```

Finalizes SHA512/256 hash.

Parameters:

- **sha512** SHA512 structure
- **hash** Output hash buffer

See: wc_Sha512_256Update

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
byte hash[WC_SHA512_256_DIGEST_SIZE];
int ret = wc_Sha512_256Final(&sha, hash);
```

```
void wc_Sha512_256Free(
    wc_Sha512 * sha
)
```

Frees SHA512/256 resources.

Parameters:

- **sha** SHA512 structure

See: wc_InitSha512_256

Return: none No returns

Example

```
wc_Sha512 sha;
wc_InitSha512_256(&sha);
wc_Sha512_256Free(&sha);
```

```
int wc_Sha512_256Reset(
    wc_Sha512 * sha
)
```

Resets the sha structure to its freshly initialized state, continuing with its established allocations, heap hint, and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha512_256Free for teardown at end of life.

Parameters:

- **sha** Pointer to the sha structure to be reset.

See:

- wc_InitSha512_256
- wc_Sha512_256Final
- wc_Sha512_256Free

Return:

- 0 Returned upon successfully resetting the structure.
- BAD_FUNC_ARG Returned if sha is NULL.
- negative Other values may be returned on reinitialization failure in fallback (non-plain-software) configurations.

Example

```
wc_Sha512 sha;
byte data[] = { Data to be discarded };
byte data2[] = { Data to be hashed };
byte hash[WC_SHA512_256_DIGEST_SIZE];

if ((ret = wc_InitSha512_256(&sha)) == 0) {
    wc_Sha512_256Update(&sha, data, sizeof(data));
    wc_Sha512_256Reset(&sha); /* discard the partially hashed message */
    wc_Sha512_256Update(&sha, data2, sizeof(data2));
    wc_Sha512_256Final(&sha, hash);
    wc_Sha512_256Free(&sha);
}
```

```
int wc_Sha512_256GetHash(
    wc_Sha512 * sha512,
    byte * hash
)
```

Gets SHA512/256 hash without finalizing.

Parameters:

- **sha512** SHA512 structure
- **hash** Output hash buffer

See: wc_Sha512_256Final

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
byte hash[WC_SHA512_256_DIGEST_SIZE];
int ret = wc_Sha512_256GetHash(&sha, hash);
```

```
int wc_Sha512_256Copy(
    wc_Sha512 * src,
    wc_Sha512 * dst
)
```

Copies SHA512/256 context.

Parameters:

- **src** Source SHA512 structure
- **dst** Destination SHA512 structure (must be zeroed or previously initialized)

See: wc_InitSha512_256

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 src, dst;
memset(&dst, 0, sizeof(dst));
int ret = wc_InitSha512_256(&src);
if (ret == 0)
    ret = wc_Sha512_256Copy(&src, &dst);
```

```
int wc_Sha512_256SetFlags(
    wc_Sha512 * sha512,
    word32 flags
)
```

Sets SHA512/256 flags.

Parameters:

- **sha512** SHA512 structure
- **flags** Flags to set

See: wc_InitSha512_256

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
int ret = wc_Sha512_256SetFlags(&sha, WC_HASH_FLAG_WILLCOPY);
```

```
int wc_Sha512_256GetFlags(
    wc_Sha512 * sha512,
    word32 * flags
)
```

Gets SHA512/256 flags.

Parameters:

- **sha512** SHA512 structure
- **flags** Pointer to store flags

See: wc_Sha512_256SetFlags

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
word32 flags;
int ret = wc_Sha512_256GetFlags(&sha, &flags);
```

```
int wc_Sha512_256Transform(
    wc_Sha512 * sha,
    const unsigned char * data
)
```

Transforms SHA512/256 block.

Parameters:

- **sha** SHA512 structure
- **data** Block data

See: wc_Sha512_256Update

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
unsigned char block[WC_SHA512_BLOCK_SIZE];
int ret = wc_Sha512_256Transform(&sha, block);
```

```
int wc_InitSha384_ex(
    wc_Sha384 * sha,
    void * heap,
    int devId
)
```

Initializes SHA384 with heap and device ID.

Parameters:

- **sha** SHA384 structure
- **heap** Heap hint
- **devId** Device ID

See: wc_InitSha384

Return:

- 0 on success
- negative on error

Example

```
wc_Sha384 sha;
int ret = wc_InitSha384_ex(&sha, NULL, INVALID_DEVID);
```

```
int wc_Sha384FinalRaw(
    wc_Sha384 * sha384,
    byte * hash
)
```

Gets raw SHA384 hash without finalizing.

Parameters:

- **sha384** SHA384 structure
- **hash** Output hash buffer

See: wc_Sha384Final

Return:

- 0 on success
- negative on error

Example

```
wc_Sha384 sha;
byte hash[WC_SHA384_DIGEST_SIZE];
int ret = wc_Sha384FinalRaw(&sha, hash);
```

```
void wc_Sha384Free(
    wc_Sha384 * sha
)
```

Frees SHA384 resources.

Parameters:

- **sha** SHA384 structure

See: wc_InitSha384

Return: none No returns

Example

```
wc_Sha384 sha;
wc_InitSha384(&sha);
wc_Sha384Free(&sha);
```

```
int wc_Sha384Reset(
    wc_Sha384 * sha
)
```

Resets the sha structure to its freshly initialized state, continuing with its established allocations, heap hint, and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha384Free for teardown at end of life.

Parameters:

- **sha** Pointer to the sha structure to be reset.

See:

- wc_InitSha384
- wc_Sha384Final
- wc_Sha384Free

Return:

- 0 Returned upon successfully resetting the structure.

- BAD_FUNC_ARG Returned if sha is NULL.
- negative Other values may be returned on reinitialization failure in fallback (non-plain-software) configurations.

Example

```
wc_Sha384 sha;
byte data[] = { Data to be discarded };
byte data2[] = { Data to be hashed };
byte hash[WC_SHA384_DIGEST_SIZE];

if ((ret = wc_InitSha384(&sha)) == 0) {
    wc_Sha384Update(&sha, data, sizeof(data));
    wc_Sha384Reset(&sha); /* discard the partially hashed message */
    wc_Sha384Update(&sha, data2, sizeof(data2));
    wc_Sha384Final(&sha, hash);
    wc_Sha384Free(&sha);
}
```

```
int wc_Sha384GetHash(
    wc_Sha384 * sha384,
    byte * hash
)
```

Gets SHA384 hash without finalizing.

Parameters:

- **sha384** SHA384 structure
- **hash** Output hash buffer

See: wc_Sha384Final

Return:

- 0 on success
- negative on error

Example

```
wc_Sha384 sha;
byte hash[WC_SHA384_DIGEST_SIZE];
int ret = wc_Sha384GetHash(&sha, hash);
```

```
int wc_Sha384Copy(
    wc_Sha384 * src,
    wc_Sha384 * dst
)
```

Copies SHA384 context.

Parameters:

- **src** Source SHA384 structure
- **dst** Destination SHA384 structure (must be zeroed or previously initialized)

See: wc_InitSha384

Return:

- 0 on success
- negative on error

Example

```
wc_Sha384 src, dst;
memset(&dst, 0, sizeof(dst));
int ret = wc_InitSha384(&src);
if (ret == 0)
    ret = wc_Sha384Copy(&src, &dst);
```

```
int wc_Sha384_Grow(
    wc_Sha384 * sha384,
    const byte * in,
    int inSz
)
```

Grows SHA384 buffer with input data.

Parameters:

- **sha384** SHA384 structure
- **in** Input data
- **inSz** Input size

See: wc_Sha384Update

Return:

- 0 on success
- negative on error

Example

```
wc_Sha384 sha;
byte data[100];
int ret = wc_Sha384_Grow(&sha, data, sizeof(data));
```

```
int wc_Sha384SetFlags(
    wc_Sha384 * sha384,
    word32 flags
)
```

Sets SHA384 flags.

Parameters:

- **sha384** SHA384 structure
- **flags** Flags to set

See: wc_InitSha384

Return:

- 0 on success
- negative on error

Example

```
wc_Sha384 sha;
int ret = wc_Sha384SetFlags(&sha, WC_HASH_FLAG_WILLCOPY);
```

```
int wc_Sha384GetFlags(
    wc_Sha384 * sha384,
    word32 * flags
)
```

Gets SHA384 flags.

Parameters:

- **sha384** SHA384 structure
- **flags** Pointer to store flags

See: wc_Sha384SetFlags

Return:

- 0 on success
- negative on error

Example

```
wc_Sha384 sha;
word32 flags;
int ret = wc_Sha384GetFlags(&sha, &flags);
```

```
int wc_Sha384Transform(
    wc_Sha384 * sha,
    const unsigned char * data
)
```

Transforms SHA384 block.

Parameters:

- **sha** SHA384 structure
- **data** Block data

See: wc_Sha384Update

Return:

- 0 on success
- negative on error

Example

```
wc_Sha384 sha;
unsigned char block[WC_SHA384_BLOCK_SIZE];
int ret = wc_Sha384Transform(&sha, block);
```

B.51 Algorithms - SHE

B.50.2.172 function wc_Sha384Transform

B.51.1 Functions

	Name
int	**wc_SHE_Init**(wc_SHE * she, void * heap, int devId)Initialize a SHE context with a heap hint and device ID.
int	**wc_SHE_Init_Id**(wc_SHE * she, unsigned char * id, int len, void * heap, int devId)Initialize a SHE context with an opaque hardware key identifier. Useful when using crypto callbacks and additional info needs to be attached to the SHE context to determine slot or key group information.
int	**wc_SHE_Init_Label**(wc_SHE * she, const char * label, void * heap, int devId)Initialize a SHE context with a human-readable key label. Useful when using crypto callbacks and additional info needs to be attached to the SHE context to determine slot or key group information.

	Name
void	**wc_SHE_Free**(wc_SHE * she)Scrub all data and zero the SHE context. Safe to call on a NULL pointer.
int	**wc_SHE_GetUID**(wc_SHE * she, byte * uid, word32 uidSz, const void * ctx)Retrieve the UID from hardware via a crypto callback. Requires WOLF_CRYPTO_CB and that NO_WC_SHE_GETUID is not defined.
int	**wc_SHE_GetCounter**(wc_SHE * she, word32 * counter, const void * ctx)Retrieve the monotonic counter value from hardware via a crypto callback. The SHE spec uses a 28-bit counter. The caller should increment this value before passing to GenerateM1M2M3 or GenerateM4M5. Requires WOLF_CRYPTO_CB and that NO_WC_SHE_GETCOUNTER is not defined.
int	**wc_SHE_SetKdfConstants**(wc_SHE * she, const byte * encC, word32 encCSz, const byte * macC, word32 macCSz)Set custom KDF constants used in Miyaguchi-Preneel key derivation. Defaults are KEY_UPDATE_ENC_C and KEY_UPDATE_MAC_C from the SHE spec. Either pointer may be NULL to leave that constant unchanged. Requires WOLFSSL_SHE_EXTENDED.
int	**wc_SHE_SetM2Header**(wc_SHE * she, const byte * header, word32 headerSz)Override the M2 cleartext header (first 16 bytes of M2 before encryption). When set, GenerateM1M2M3 uses this instead of auto-building from counter and flags. Requires WOLFSSL_SHE_EXTENDED.
int	**wc_SHE_SetM4Header**(wc_SHE * she, const byte * header, word32 headerSz)Override the M4 cleartext counter block (16_byte block encrypted with K3). When set, GenerateM4M5 uses this instead of auto-building from counter. Requires WOLFSSL_SHE_EXTENDED.
int	**wc_SHE_ImportM1M2M3**(wc_SHE * she, const byte * m1, word32 m1Sz, const byte * m2, word32 m2Sz, const byte * m3, word32 m3Sz)Import externally-provided M1/M2/M3 into the SHE context. Sets the generated flag so the callback for GenerateM4M5 can read M1/M2/M3 from the context to send to hardware. Requires WOLF_CRYPTO_CB and that NO_WC_SHE_IMPORT_M123 is not defined.

	Name
int	**wc_SHE_GenerateM1M2M3**(wc_SHE * she, const byte * uid, word32 uidSz, byte authKeyId, const byte * authKey, word32 authKeySz, byte targetKeyId, const byte * newKey, word32 newKeySz, word32 counter, byte flags, byte * m1, word32 m1Sz, byte * m2, word32 m2Sz, byte * m3, word32 m3Sz)Generate SHE key update messages M1, M2, and M3 and write them to caller_provided buffers. Uses Miyaguchi_Preneel AES_128 KDF to derive K1 and K2 from the authorizing key, AES_CBC to encrypt the new key (M2), and AES_CMAC for authentication (M3).
int	**wc_SHE_GenerateM4M5**(wc_SHE * she, const byte * uid, word32 uidSz, byte authKeyId, byte targetKeyId, const byte * newKey, word32 newKeySz, word32 counter, byte * m4, word32 m4Sz, byte * m5, word32 m5Sz)Generate SHE verification messages M4 and M5 and write them to caller-provided buffers. Uses Miyaguchi-Preneel AES-128 KDF to derive K3 and K4 from the new key, AES-ECB for the M4 counter block, and AES-CMAC for M5. Independent of M1/M2/M3 and can be called on a separate context.
int	**wc_SHE_LoadKey**(void * heap, int devId, const byte * m1, word32 m1Sz, const byte * m2, word32 m2Sz, const byte * m3, word32 m3Sz, byte * m4, word32 m4Sz, byte * m5, word32 m5Sz)One_shot convenience wrapper: Init, ImportM1M2M3, GenerateM4M5 (via callback), Free. Dispatches to a hardware crypto callback that sends M1/M2/M3 to the HSM and returns M4/M5. Requires a valid devId (not INVALID_DEVID). Define NO_WC_SHE_LOADKEY to compile out.
int	**wc_SHE_LoadKey_Id**(unsigned char * id, int idLen, void * heap, int devId, const byte * m1, word32 m1Sz, const byte * m2, word32 m2Sz, const byte * m3, word32 m3Sz, byte * m4, word32 m4Sz, byte * m5, word32 m5Sz)One-shot Load Key with an opaque hardware key identifier. Same as wc_SHE_LoadKey but initializes the context with wc_SHE_Init_Id. Define NO_WC_SHE_LOADKEY to compile out.

	Name
int	**wc_SHE_LoadKey_Label**(const char * label, void * heap, int devId, const byte * m1, word32 m1Sz, const byte * m2, word32 m2Sz, const byte * m3, word32 m3Sz, byte * m4, word32 m4Sz, byte * m5, word32 m5Sz)One-shot Load Key with a human-readable key label. Same as wc_SHE_LoadKey but initializes the context with wc_SHE_Init_Label. Define NO_WC_SHE_LOADKEY to compile out.
int	**wc_SHE_LoadKey_Verify**(void * heap, int devId, const byte * m1, word32 m1Sz, const byte * m2, word32 m2Sz, const byte * m3, word32 m3Sz, byte * m4, word32 m4Sz, byte * m5, word32 m5Sz, const byte * m4Expected, word32 m4ExpectedSz, const byte * m5Expected, word32 m5ExpectedSz)One-shot Load Key with M4/M5 verification. Same as wc_SHE_LoadKey but also compares the M4/M5 returned by the HSM against caller-provided expected values using constant-time comparison. Returns SIG_VERIFY_E on mismatch. The actual M4/M5 are still written to the output buffers on failure. Define NO_WC_SHE_LOADKEY to compile out.
int	**wc_SHE_LoadKey_Verify_Id**(unsigned char * id, int idLen, void * heap, int devId, const byte * m1, word32 m1Sz, const byte * m2, word32 m2Sz, const byte * m3, word32 m3Sz, byte * m4, word32 m4Sz, byte * m5, word32 m5Sz, const byte * m4Expected, word32 m4ExpectedSz, const byte * m5Expected, word32 m5ExpectedSz)One-shot Load Key with opaque key identifier and M4/M5 verification. Combines wc_SHE_LoadKey_Id and wc_SHE_LoadKey_Verify. Define NO_WC_SHE_LOADKEY to compile out.
int	**wc_SHE_LoadKey_Verify_Label**(const char * label, void * heap, int devId, const byte * m1, word32 m1Sz, const byte * m2, word32 m2Sz, const byte * m3, word32 m3Sz, byte * m4, word32 m4Sz, byte * m5, word32 m5Sz, const byte * m4Expected, word32 m4ExpectedSz, const byte * m5Expected, word32 m5ExpectedSz)One-shot Load Key with key label and M4/M5 verification. Combines wc_SHE_LoadKey_Label and wc_SHE_LoadKey_Verify. Define NO_WC_SHE_LOADKEY to compile out.

	Name
int	**wc_SHE_ExportKey**(wc_SHE * she, byte * m1, word32 m1Sz, byte * m2, word32 m2Sz, byte * m3, word32 m3Sz, byte * m4, word32 m4Sz, byte * m5, word32 m5Sz, const void * ctx)Export a key from hardware in SHE loadable format (M1_M5). Some HSMs allow exporting certain key slots (e.g. RAM key) so they can be re-loaded later via the SHE key update protocol. Requires WOLF_CRYPTO_CB and that NO_WC_SHE_EXPORTKEY is not defined. Any output buffer may be NULL to skip that message.
int	**wc_SHE_AesMp16**(Aes * aes, const byte * in, word32 inSz, byte * out)Miyaguchi_Preneel AES_128 one_way compression function. H_0 = 0, H_i = E_{H_{i_1}}(M_i) XOR M_i XOR H_{i-1}. Only valid for AES-128 where key size equals block size. This is an internal function exposed for testing purposes.

B.51.2 Functions Documentation

```
int wc_SHE_Init(
    wc_SHE * she,
    void * heap,
    int devId
)
```

Initialize a SHE context with a heap hint and device ID.

Parameters:

- **she** pointer to a wc_SHE structure to initialize
- **heap** heap hint for internal allocations, or NULL
- **devId** crypto callback device ID, or INVALID_DEVID for software-only operation

See:

- wc_SHE_Init_Id
- wc_SHE_Init_Label
- wc_SHE_Free

Return:

- 0 on success
- BAD_FUNC_ARG if she is NULL

Example

```
wc_SHE she;
int ret;
ret = wc_SHE_Init(&she, NULL, INVALID_DEVID);
if (ret == 0) {
    // use she context
}
wc_SHE_Free(&she);
```

```
int wc_SHE_Init_Id(
    wc_SHE * she,
    unsigned char * id,
    int len,
    void * heap,
    int devId
)
```

Initialize a SHE context with an opaque hardware key identifier. Useful when using crypto callbacks and additional info needs to be attached to the SHE context to determine slot or key group information.

Parameters:

- **she** pointer to a wc_SHE structure to initialize
- **id** opaque key identifier bytes
- **len** length of id in bytes (0 to WC_SHE_MAX_ID_LEN)
- **heap** heap hint for internal allocations, or NULL
- **devId** crypto callback device ID

See:

- wc_SHE_Init
- wc_SHE_Init_Label
- wc_SHE_Free

Return:

- 0 on success
- BAD_FUNC_ARG if she is NULL, id is NULL when len > 0, or len exceeds WC_SHE_MAX_ID_LEN

Example

```
wc_SHE she;
unsigned char myId[] = { 0x01, 0x02, 0x03 };
int ret;
ret = wc_SHE_Init_Id(&she, myId, sizeof(myId), NULL, myDevId);
```

```
int wc_SHE_Init_Label(
    wc_SHE * she,
    const char * label,
```

```
    void * heap,
    int devId
)
```

Initialize a SHE context with a human-readable key label. Useful when using crypto callbacks and additional info needs to be attached to the SHE context to determine slot or key group information.

Parameters:

- **she** pointer to a wc_SHE structure to initialize
- **label** NUL-terminated key label string
- **heap** heap hint for internal allocations, or NULL
- **devId** crypto callback device ID

See:

- wc_SHE_Init
- wc_SHE_Init_Id
- wc_SHE_Free

Return:

- 0 on success
- BAD_FUNC_ARG if she or label is NULL, or label length exceeds WC_SHE_MAX_LABEL_LEN

Example

```
wc_SHE she;
int ret;
ret = wc_SHE_Init_Label(&she, "ecu-master-key", NULL, myDevId);
```

```
void wc_SHE_Free(
    wc_SHE * she
)
```

Scrub all data and zero the SHE context. Safe to call on a NULL pointer.

Parameters:

- **she** pointer to a wc_SHE structure, or NULL

See: wc_SHE_Init

Example

```
wc_SHE she;
wc_SHE_Init(&she, NULL, INVALID_DEVID);
// ... use context ...
wc_SHE_Free(&she);
```

```
int wc_SHE_GetUID(
    wc_SHE * she,
    byte * uid,
    word32 uidSz,
    const void * ctx
)
```

Retrieve the UID from hardware via a crypto callback. Requires WOLF_CRYPTO_CB and that NO_WC_SHE_GETUID is not defined.

Parameters:

- **she** initialized SHE context
- **uid** buffer to receive the 15-byte (120-bit) SHE UID
- **uidSz** size of uid buffer in bytes (must be >= WC_SHE_UID_SZ)
- **ctx** read-only caller context passed to the callback (e.g. challenge buffer, HSM handle)

See: wc_SHE_GetCounter

Return:

- 0 on success
- BAD_FUNC_ARG if she, uid, or uidSz is invalid
- CRYPTOCB_UNAVAILABLE if no callback is registered

Example

```
byte uid[WC_SHE_UID_SZ];
int ret;
ret = wc_SHE_GetUID(&she, uid, sizeof(uid), NULL);
```

```
int wc_SHE_GetCounter(
    wc_SHE * she,
    word32 * counter,
    const void * ctx
)
```

Retrieve the monotonic counter value from hardware via a crypto callback. The SHE spec uses a 28-bit counter. The caller should increment this value before passing to GenerateM1M2M3 or GenerateM4M5. Requires WOLF_CRYPTO_CB and that NO_WC_SHE_GETCOUNTER is not defined.

Parameters:

- **she** initialized SHE context
- **counter** pointer to receive the current counter value
- **ctx** read-only caller context passed to the callback

See: wc_SHE_GetUID

Return:

- 0 on success
- BAD_FUNC_ARG if she or counter is NULL
- CRYPTOCB_UNAVAILABLE if no callback is registered

Example

```
word32 counter;
int ret;
ret = wc_SHE_GetCounter(&she, &counter, NULL);
```

```
int wc_SHE_SetKdfConstants(
    wc_SHE * she,
    const byte * encC,
    word32 encCSz,
    const byte * macC,
    word32 macCSz
)
```

Set custom KDF constants used in Miyaguchi-Preneel key derivation. Defaults are KEY_UPDATE_ENC_C and KEY_UPDATE_MAC_C from the SHE spec. Either pointer may be NULL to leave that constant unchanged. Requires WOLFSSL_SHE_EXTENDED.

Parameters:

- **she** initialized SHE context
- **encC** 16-byte encryption derivation constant (CENC), or NULL
- **encCSz** must be WC_SHE_KEY_SZ (16) when encC is non-NULL
- **macC** 16-byte MAC derivation constant (CMAC), or NULL
- **macCSz** must be WC_SHE_KEY_SZ (16) when macC is non-NULL

See:

- wc_SHE_SetM2Header
- wc_SHE_SetM4Header
- wc_SHE_GenerateM1M2M3

Return:

- 0 on success
- BAD_FUNC_ARG if she is NULL or sizes are not WC_SHE_KEY_SZ when corresponding pointer is non-NULL

Example

```
byte myEncC[WC_SHE_KEY_SZ] = { ... };
byte myMacC[WC_SHE_KEY_SZ] = { ... };
int ret;
ret = wc_SHE_SetKdfConstants(&she, myEncC, WC_SHE_KEY_SZ,
                             myMacC, WC_SHE_KEY_SZ);
```

```
int wc_SHE_SetM2Header(
    wc_SHE * she,
    const byte * header,
    word32 headerSz
)
```

Override the M2 cleartext header (first 16 bytes of M2 before encryption). When set, GenerateM1M2M3 uses this instead of auto-building from counter and flags. Requires WOLFSSL_SHE_EXTENDED.

Parameters:

- **she** initialized SHE context
- **header** 16-byte cleartext header block
- **headerSz** must be WC_SHE_KEY_SZ (16)

See:

- wc_SHE_SetKdfConstants
- wc_SHE_SetM4Header
- wc_SHE_GenerateM1M2M3

Return:

- 0 on success
- BAD_FUNC_ARG if she or header is NULL, or headerSz is not WC_SHE_KEY_SZ

Example

```
byte header[WC_SHE_KEY_SZ] = { ... };
int ret;
ret = wc_SHE_SetM2Header(&she, header, WC_SHE_KEY_SZ);
```

```
int wc_SHE_SetM4Header(
    wc_SHE * she,
    const byte * header,
    word32 headerSz
)
```

Override the M4 cleartext counter block (16-byte block encrypted with K3). When set, GenerateM4M5 uses this instead of auto-building from counter. Requires WOLFSSL_SHE_EXTENDED.

Parameters:

- **she** initialized SHE context
- **header** 16-byte cleartext counter block
- **headerSz** must be WC_SHE_KEY_SZ (16)

See:

- wc_SHE_SetKdfConstants
- wc_SHE_SetM2Header
- wc_SHE_GenerateM4M5

Return:

- 0 on success
- BAD_FUNC_ARG if she or header is NULL, or headerSz is not WC_SHE_KEY_SZ

Example

```
byte header[WC_SHE_KEY_SZ] = { ... };
int ret;
ret = wc_SHE_SetM4Header(&she, header, WC_SHE_KEY_SZ);
```

```
int wc_SHE_ImportM1M2M3(
    wc_SHE * she,
    const byte * m1,
    word32 m1Sz,
    const byte * m2,
    word32 m2Sz,
    const byte * m3,
    word32 m3Sz
)
```

Import externally-provided M1/M2/M3 into the SHE context. Sets the generated flag so the callback for GenerateM4M5 can read M1/M2/M3 from the context to send to hardware. Requires WOLF_CRYPTO_CB and that NO_WC_SHE_IMPORT_M123 is not defined.

Parameters:

- **she** initialized SHE context
- **m1** 16-byte M1 message (UID | KeyID | AuthID)
- **m1Sz** must be WC_SHE_M1_SZ (16)
- **m2** 32-byte M2 message (encrypted counter|flags|pad|newkey)
- **m2Sz** must be WC_SHE_M2_SZ (32)
- **m3** 16-byte M3 message (CMAC over M1|M2)
- **m3Sz** must be WC_SHE_M3_SZ (16)

See:

- wc_SHE_GenerateM1M2M3
- wc_SHE_GenerateM4M5
- wc_SHE_LoadKey

Return:

- 0 on success
- BAD_FUNC_ARG if she is NULL or any message size is incorrect

Example

```
int ret;
ret = wc_SHE_ImportM1M2M3(&she,
          m1, WC_SHE_M1_SZ,
          m2, WC_SHE_M2_SZ,
          m3, WC_SHE_M3_SZ);
```

```
int wc_SHE_GenerateM1M2M3(
    wc_SHE * she,
    const byte * uid,
    word32 uidSz,
    byte authKeyId,
    const byte * authKey,
    word32 authKeySz,
    byte targetKeyId,
    const byte * newKey,
    word32 newKeySz,
    word32 counter,
    byte flags,
    byte * m1,
    word32 m1Sz,
    byte * m2,
    word32 m2Sz,
    byte * m3,
    word32 m3Sz
)
```

Generate SHE key update messages M1, M2, and M3 and write them to caller-provided buffers. Uses Miyaguchi-Preneel AES-128 KDF to derive K1 and K2 from the authorizing key, AES-CBC to encrypt the new key (M2), and AES-CMAC for authentication (M3).

Parameters:

- **she** initialized SHE context
- **uid** 15-byte SHE UID (120-bit ECU/module identifier)
- **uidSz** must be WC_SHE_UID_SZ (15)
- **authKeyId** 4-bit slot ID of the authorizing key
- **authKey** 16-byte value of the authorizing key
- **authKeySz** must be WC_SHE_KEY_SZ (16)
- **targetKeyId** 4-bit slot ID of the key being loaded
- **newKey** 16-byte value of the new key to load
- **newKeySz** must be WC_SHE_KEY_SZ (16)
- **counter** 28-bit monotonic counter value (must be strictly greater than the counter stored in the target slot)
- **flags** key protection flags (lower 4 bits)
- **m1** output buffer for M1 (16 bytes)
- **m1Sz** size of m1 buffer, must be >= WC_SHE_M1_SZ
- **m2** output buffer for M2 (32 bytes)
- **m2Sz** size of m2 buffer, must be >= WC_SHE_M2_SZ
- **m3** output buffer for M3 (16 bytes)

- **m3Sz** size of m3 buffer, must be >= WC_SHE_M3_SZ

See:

- wc_SHE_GenerateM4M5
- wc_SHE_ImportM1M2M3
- wc_SHE_LoadKey

Return:

- 0 on success
- BAD_FUNC_ARG if any required pointer is NULL, sizes are incorrect, or counter, flags, authKeyId or targetKeyId is wider than its packed field

Example

```
byte m1[WC_SHE_M1_SZ], m2[WC_SHE_M2_SZ], m3[WC_SHE_M3_SZ];
int ret;
ret = wc_SHE_GenerateM1M2M3(&she,
          uid, WC_SHE_UID_SZ,
          authKeyId, authKey, WC_SHE_KEY_SZ,
          targetKeyId, newKey, WC_SHE_KEY_SZ,
          counter, flags,
          m1, WC_SHE_M1_SZ,
          m2, WC_SHE_M2_SZ,
          m3, WC_SHE_M3_SZ);
```

```
int wc_SHE_GenerateM4M5(
    wc_SHE * she,
    const byte * uid,
    word32 uidSz,
    byte authKeyId,
    byte targetKeyId,
    const byte * newKey,
    word32 newKeySz,
    word32 counter,
    byte * m4,
    word32 m4Sz,
    byte * m5,
    word32 m5Sz
)
```

Generate SHE verification messages M4 and M5 and write them to caller-provided buffers. Uses Miyaguchi-Preneel AES-128 KDF to derive K3 and K4 from the new key, AES-ECB for the M4 counter block, and AES-CMAC for M5. Independent of M1/M2/M3 and can be called on a separate context.

Parameters:

- **she** initialized SHE context
- **uid** 15-byte SHE UID (same UID used for M1)

- **uidSz** must be WC_SHE_UID_SZ (15)
- **authKeyId** 4-bit slot ID of the authorizing key (same as in M1)
- **targetKeyId** 4-bit slot ID of the key being loaded (same as in M1)
- **newKey** 16-byte value of the new key
- **newKeySz** must be WC_SHE_KEY_SZ (16)
- **counter** 28-bit monotonic counter (same value as in M2)
- **m4** output buffer for M4 (32 bytes)
- **m4Sz** size of m4 buffer, must be >= WC_SHE_M4_SZ
- **m5** output buffer for M5 (16 bytes)
- **m5Sz** size of m5 buffer, must be >= WC_SHE_M5_SZ

See:

- wc_SHE_GenerateM1M2M3
- wc_SHE_LoadKey_Verify

Return:

- 0 on success
- BAD_FUNC_ARG if any required pointer is NULL, sizes are incorrect, or counter, authKeyId or targetKeyId is wider than its packed field

Example

```
byte m4[WC_SHE_M4_SZ], m5[WC_SHE_M5_SZ];
int ret;
ret = wc_SHE_GenerateM4M5(&she,
          uid, WC_SHE_UID_SZ,
          authKeyId, targetKeyId,
          newKey, WC_SHE_KEY_SZ,
          counter,
          m4, WC_SHE_M4_SZ,
          m5, WC_SHE_M5_SZ);
```

```
int wc_SHE_LoadKey(
    void * heap,
    int devId,
    const byte * m1,
    word32 m1Sz,
    const byte * m2,
    word32 m2Sz,
    const byte * m3,
    word32 m3Sz,
    byte * m4,
    word32 m4Sz,
    byte * m5,
    word32 m5Sz
)
```

One-shot convenience wrapper: Init, ImportM1M2M3, GenerateM4M5 (via callback), Free. Dispatches to a hardware crypto callback that sends M1/M2/M3 to the HSM and returns M4/M5. Requires a valid devId (not INVALID_DEVID). Define NO_WC_SHE_LOADKEY to compile out.

Parameters:

- **heap** heap hint for internal allocations, or NULL
- **devId** crypto callback device ID (must not be INVALID_DEVID)
- **m1** 16-byte M1 input message
- **m1Sz** must be WC_SHE_M1_SZ (16)
- **m2** 32-byte M2 input message
- **m2Sz** must be WC_SHE_M2_SZ (32)
- **m3** 16-byte M3 input message
- **m3Sz** must be WC_SHE_M3_SZ (16)
- **m4** output buffer for M4 (32 bytes)
- **m4Sz** size of m4 buffer, must be >= WC_SHE_M4_SZ
- **m5** output buffer for M5 (16 bytes)
- **m5Sz** size of m5 buffer, must be >= WC_SHE_M5_SZ

See:

- wc_SHE_LoadKey_Id
- wc_SHE_LoadKey_Label
- wc_SHE_LoadKey_Verify
- wc_SHE_ImportM1M2M3
- wc_SHE_GenerateM4M5

Return:

- 0 on success
- BAD_FUNC_ARG if any required pointer is NULL or sizes are incorrect

Example

```
byte m4[WC_SHE_M4_SZ], m5[WC_SHE_M5_SZ];
int ret;
ret = wc_SHE_LoadKey(NULL, myDevId,
          m1, WC_SHE_M1_SZ,
          m2, WC_SHE_M2_SZ,
          m3, WC_SHE_M3_SZ,
          m4, WC_SHE_M4_SZ,
          m5, WC_SHE_M5_SZ);
```

```
int wc_SHE_LoadKey_Id(
    unsigned char * id,
    int idLen,
    void * heap,
    int devId,
    const byte * m1,
    word32 m1Sz,
```

```
    const byte * m2,
    word32 m2Sz,
    const byte * m3,
    word32 m3Sz,
    byte * m4,
    word32 m4Sz,
    byte * m5,
    word32 m5Sz
)
```

One-shot Load Key with an opaque hardware key identifier. Same as wc_SHE_LoadKey but initializes the context with wc_SHE_Init_Id. Define NO_WC_SHE_LOADKEY to compile out.

Parameters:

- **id** opaque key identifier bytes
- **idLen** length of id in bytes
- **heap** heap hint for internal allocations, or NULL
- **devId** crypto callback device ID (must not be INVALID_DEVID)
- **m1** 16-byte M1 input message
- **m1Sz** must be WC_SHE_M1_SZ (16)
- **m2** 32-byte M2 input message
- **m2Sz** must be WC_SHE_M2_SZ (32)
- **m3** 16-byte M3 input message
- **m3Sz** must be WC_SHE_M3_SZ (16)
- **m4** output buffer for M4 (32 bytes)
- **m4Sz** size of m4 buffer, must be >= WC_SHE_M4_SZ
- **m5** output buffer for M5 (16 bytes)
- **m5Sz** size of m5 buffer, must be >= WC_SHE_M5_SZ

See:

- wc_SHE_LoadKey
- wc_SHE_LoadKey_Label
- wc_SHE_LoadKey_Verify_Id

Return:

- 0 on success
- BAD_FUNC_ARG if any required pointer is NULL or sizes are incorrect

Example

```
byte m4[WC_SHE_M4_SZ], m5[WC_SHE_M5_SZ];
unsigned char keyId[] = { 0x01, 0x02 };
int ret;
ret = wc_SHE_LoadKey_Id(keyId, sizeof(keyId), NULL, myDevId,
          m1, WC_SHE_M1_SZ,
          m2, WC_SHE_M2_SZ,
          m3, WC_SHE_M3_SZ,
          m4, WC_SHE_M4_SZ,
          m5, WC_SHE_M5_SZ);
```

```
int wc_SHE_LoadKey_Label(
    const char * label,
    void * heap,
    int devId,
    const byte * m1,
    word32 m1Sz,
    const byte * m2,
    word32 m2Sz,
    const byte * m3,
    word32 m3Sz,
    byte * m4,
    word32 m4Sz,
    byte * m5,
    word32 m5Sz
)
```

One-shot Load Key with a human-readable key label. Same as wc_SHE_LoadKey but initializes the context with wc_SHE_Init_Label. Define NO_WC_SHE_LOADKEY to compile out.

Parameters:

- **label** NUL-terminated key label string
- **heap** heap hint for internal allocations, or NULL
- **devId** crypto callback device ID (must not be INVALID_DEVID)
- **m1** 16-byte M1 input message
- **m1Sz** must be WC_SHE_M1_SZ (16)
- **m2** 32-byte M2 input message
- **m2Sz** must be WC_SHE_M2_SZ (32)
- **m3** 16-byte M3 input message
- **m3Sz** must be WC_SHE_M3_SZ (16)
- **m4** output buffer for M4 (32 bytes)
- **m4Sz** size of m4 buffer, must be >= WC_SHE_M4_SZ
- **m5** output buffer for M5 (16 bytes)
- **m5Sz** size of m5 buffer, must be >= WC_SHE_M5_SZ

See:

- wc_SHE_LoadKey
- wc_SHE_LoadKey_Id
- wc_SHE_LoadKey_Verify_Label

Return:

- 0 on success
- BAD_FUNC_ARG if any required pointer is NULL or sizes are incorrect

Example

```
byte m4[WC_SHE_M4_SZ], m5[WC_SHE_M5_SZ];
int ret;
```

```
ret = wc_SHE_LoadKey_Label("ecu-master", NULL, myDevId,
          m1, WC_SHE_M1_SZ,
          m2, WC_SHE_M2_SZ,
          m3, WC_SHE_M3_SZ,
          m4, WC_SHE_M4_SZ,
          m5, WC_SHE_M5_SZ);
```

```
int wc_SHE_LoadKey_Verify(
    void * heap,
    int devId,
    const byte * m1,
    word32 m1Sz,
    const byte * m2,
    word32 m2Sz,
    const byte * m3,
    word32 m3Sz,
    byte * m4,
    word32 m4Sz,
    byte * m5,
    word32 m5Sz,
    const byte * m4Expected,
    word32 m4ExpectedSz,
    const byte * m5Expected,
    word32 m5ExpectedSz
)
```

One-shot Load Key with M4/M5 verification. Same as wc_SHE_LoadKey but also compares the M4/M5 returned by the HSM against caller-provided expected values using constant-time comparison. Returns SIG_VERIFY_E on mismatch. The actual M4/M5 are still written to the output buffers on failure. Define NO_WC_SHE_LOADKEY to compile out.

Parameters:

- **heap** heap hint for internal allocations, or NULL
- **devId** crypto callback device ID (must not be INVALID_DEVID)
- **m1** 16-byte M1 input message
- **m1Sz** must be WC_SHE_M1_SZ (16)
- **m2** 32-byte M2 input message
- **m2Sz** must be WC_SHE_M2_SZ (32)
- **m3** 16-byte M3 input message
- **m3Sz** must be WC_SHE_M3_SZ (16)
- **m4** output buffer for M4 (32 bytes)
- **m4Sz** size of m4 buffer, must be >= WC_SHE_M4_SZ
- **m5** output buffer for M5 (16 bytes)
- **m5Sz** size of m5 buffer, must be >= WC_SHE_M5_SZ
- **m4Expected** expected M4 verification message to compare against
- **m4ExpectedSz** must be WC_SHE_M4_SZ (32)
- **m5Expected** expected M5 verification message to compare against
- **m5ExpectedSz** must be WC_SHE_M5_SZ (16)

See:

- wc_SHE_LoadKey
- wc_SHE_LoadKey_Verify_Id
- wc_SHE_LoadKey_Verify_Label

Return:

- 0 on success
- SIG_VERIFY_E if M4/M5 do not match expected values
- BAD_FUNC_ARG if any required pointer is NULL or sizes are incorrect

Example

```
byte m4[WC_SHE_M4_SZ], m5[WC_SHE_M5_SZ];
int ret;
ret = wc_SHE_LoadKey_Verify(NULL, myDevId,
          m1, WC_SHE_M1_SZ,
          m2, WC_SHE_M2_SZ,
          m3, WC_SHE_M3_SZ,
          m4, WC_SHE_M4_SZ,
          m5, WC_SHE_M5_SZ,
          expectedM4, WC_SHE_M4_SZ,
          expectedM5, WC_SHE_M5_SZ);
if (ret == SIG_VERIFY_E) {
    // M4/M5 mismatch
}
```

```
int wc_SHE_LoadKey_Verify_Id(
    unsigned char * id,
    int idLen,
    void * heap,
    int devId,
    const byte * m1,
    word32 m1Sz,
    const byte * m2,
    word32 m2Sz,
    const byte * m3,
    word32 m3Sz,
    byte * m4,
    word32 m4Sz,
    byte * m5,
    word32 m5Sz,
    const byte * m4Expected,
    word32 m4ExpectedSz,
    const byte * m5Expected,
    word32 m5ExpectedSz
)
```

One-shot Load Key with opaque key identifier and M4/M5 verification. Combines wc_SHE_LoadKey_Id and wc_SHE_LoadKey_Verify. Define NO_WC_SHE_LOADKEY to compile out.

Parameters:

- **id** opaque key identifier bytes
- **idLen** length of id in bytes
- **heap** heap hint for internal allocations, or NULL
- **devId** crypto callback device ID (must not be INVALID_DEVID)
- **m1** 16-byte M1 input message
- **m1Sz** must be WC_SHE_M1_SZ (16)
- **m2** 32-byte M2 input message
- **m2Sz** must be WC_SHE_M2_SZ (32)
- **m3** 16-byte M3 input message
- **m3Sz** must be WC_SHE_M3_SZ (16)
- **m4** output buffer for M4 (32 bytes)
- **m4Sz** size of m4 buffer, must be >= WC_SHE_M4_SZ
- **m5** output buffer for M5 (16 bytes)
- **m5Sz** size of m5 buffer, must be >= WC_SHE_M5_SZ
- **m4Expected** expected M4 verification message
- **m4ExpectedSz** must be WC_SHE_M4_SZ (32)
- **m5Expected** expected M5 verification message
- **m5ExpectedSz** must be WC_SHE_M5_SZ (16)

See:

- wc_SHE_LoadKey_Id
- wc_SHE_LoadKey_Verify
- wc_SHE_LoadKey_Verify_Label

Return:

- 0 on success
- SIG_VERIFY_E if M4/M5 do not match expected values
- BAD_FUNC_ARG if any required pointer is NULL or sizes are incorrect

Example

```
byte m4[WC_SHE_M4_SZ], m5[WC_SHE_M5_SZ];
unsigned char keyId[] = { 0x01, 0x02 };
int ret;
ret = wc_SHE_LoadKey_Verify_Id(keyId, sizeof(keyId), NULL, myDevId,
          m1, WC_SHE_M1_SZ, m2, WC_SHE_M2_SZ, m3, WC_SHE_M3_SZ,
          m4, WC_SHE_M4_SZ, m5, WC_SHE_M5_SZ,
          expectedM4, WC_SHE_M4_SZ, expectedM5, WC_SHE_M5_SZ);
```

```
int wc_SHE_LoadKey_Verify_Label(
    const char * label,
    void * heap,
    int devId,
    const byte * m1,
    word32 m1Sz,
    const byte * m2,
    word32 m2Sz,
```

```
    const byte * m3,
    word32 m3Sz,
    byte * m4,
    word32 m4Sz,
    byte * m5,
    word32 m5Sz,
    const byte * m4Expected,
    word32 m4ExpectedSz,
    const byte * m5Expected,
    word32 m5ExpectedSz
)
```

One-shot Load Key with key label and M4/M5 verification. Combines wc_SHE_LoadKey_Label and wc_SHE_LoadKey_Verify. Define NO_WC_SHE_LOADKEY to compile out.

Parameters:

- **label** NUL-terminated key label string
- **heap** heap hint for internal allocations, or NULL
- **devId** crypto callback device ID (must not be INVALID_DEVID)
- **m1** 16-byte M1 input message
- **m1Sz** must be WC_SHE_M1_SZ (16)
- **m2** 32-byte M2 input message
- **m2Sz** must be WC_SHE_M2_SZ (32)
- **m3** 16-byte M3 input message
- **m3Sz** must be WC_SHE_M3_SZ (16)
- **m4** output buffer for M4 (32 bytes)
- **m4Sz** size of m4 buffer, must be >= WC_SHE_M4_SZ
- **m5** output buffer for M5 (16 bytes)
- **m5Sz** size of m5 buffer, must be >= WC_SHE_M5_SZ
- **m4Expected** expected M4 verification message
- **m4ExpectedSz** must be WC_SHE_M4_SZ (32)
- **m5Expected** expected M5 verification message
- **m5ExpectedSz** must be WC_SHE_M5_SZ (16)

See:

- wc_SHE_LoadKey_Label
- wc_SHE_LoadKey_Verify
- wc_SHE_LoadKey_Verify_Id

Return:

- 0 on success
- SIG_VERIFY_E if M4/M5 do not match expected values
- BAD_FUNC_ARG if any required pointer is NULL or sizes are incorrect

Example

```
byte m4[WC_SHE_M4_SZ], m5[WC_SHE_M5_SZ];
int ret;
ret = wc_SHE_LoadKey_Verify_Label("ecu-master", NULL, myDevId,
```

```
        m1, WC_SHE_M1_SZ, m2, WC_SHE_M2_SZ, m3, WC_SHE_M3_SZ,
        m4, WC_SHE_M4_SZ, m5, WC_SHE_M5_SZ,
        expectedM4, WC_SHE_M4_SZ, expectedM5, WC_SHE_M5_SZ);
```

```
int wc_SHE_ExportKey(
    wc_SHE * she,
    byte * m1,
    word32 m1Sz,
    byte * m2,
    word32 m2Sz,
    byte * m3,
    word32 m3Sz,
    byte * m4,
    word32 m4Sz,
    byte * m5,
    word32 m5Sz,
    const void * ctx
)
```

Export a key from hardware in SHE loadable format (M1-M5). Some HSMs allow exporting certain key slots (e.g. RAM key) so they can be re-loaded later via the SHE key update protocol. Requires WOLF_CRYPTO_CB and that NO_WC_SHE_EXPORTKEY is not defined. Any output buffer may be NULL to skip that message.

Parameters:

- **she** initialized SHE context
- **m1** output buffer for M1 (16 bytes), or NULL to skip
- **m1Sz** size of m1 buffer
- **m2** output buffer for M2 (32 bytes), or NULL to skip
- **m2Sz** size of m2 buffer
- **m3** output buffer for M3 (16 bytes), or NULL to skip
- **m3Sz** size of m3 buffer
- **m4** output buffer for M4 (32 bytes), or NULL to skip
- **m4Sz** size of m4 buffer
- **m5** output buffer for M5 (16 bytes), or NULL to skip
- **m5Sz** size of m5 buffer
- **ctx** read-only caller context passed to the callback

See:

- wc_SHE_ImportM1M2M3
- wc_SHE_LoadKey

Return:

- 0 on success
- BAD_FUNC_ARG if she is NULL
- CRYPTOCB_UNAVAILABLE if no callback is registered

Example

```
byte m1[WC_SHE_M1_SZ], m2[WC_SHE_M2_SZ], m3[WC_SHE_M3_SZ];
byte m4[WC_SHE_M4_SZ], m5[WC_SHE_M5_SZ];
int ret;
ret = wc_SHE_ExportKey(&she,
          m1, WC_SHE_M1_SZ,
          m2, WC_SHE_M2_SZ,
          m3, WC_SHE_M3_SZ,
          m4, WC_SHE_M4_SZ,
          m5, WC_SHE_M5_SZ,
          NULL);
```

```
int wc_SHE_AesMp16(
    Aes * aes,
    const byte * in,
    word32 inSz,
    byte * out
)
```

Miyaguchi-Preneel AES-128 one-way compression function. H_0 = 0, H_i = E_{H_{i-1}}(M_i) XOR M_i XOR H_{i-1}. Only valid for AES-128 where key size equals block size. This is an internal function exposed for testing purposes.

Parameters:

- **aes** caller-owned, already-initialized Aes structure
- **in** input data (e.g. BaseKey || KDF_Constant, 32 bytes)
- **inSz** length of input in bytes (zero-padded to block boundary)
- **out** output buffer for 16-byte compressed result

See:

- wc_SHE_GenerateM1M2M3
- wc_SHE_GenerateM4M5

Return:

- 0 on success
- BAD_FUNC_ARG if any pointer is NULL

Example

```
Aes aes;
byte input[32] = { ... };
byte output[WC_SHE_KEY_SZ];
int ret;
wc_AesInit(&aes, NULL, INVALID_DEVID);
ret = wc_SHE_AesMp16(&aes, input, sizeof(input), output);
wc_AesFree(&aes);
```

B.52 Algorithm - SipHash

B.51.2.20 function wc_SHE_AesMp16

B.52.1 Functions

	Name
int	**wc_InitSipHash**(SipHash * siphash, const unsigned char * key, unsigned char outSz)This function initializes SipHash with a key for a MAC size.
int	**wc_SipHashUpdate**(SipHash * siphash, const unsigned char * in, word32 inSz)Can be called to continually hash the provided byte array of length len.
int	**wc_SipHashFinal**(SipHash * siphash, unsigned char * out, unsigned char outSz)Finalizes MACing of data. Result is placed into out.
int	**wc_SipHash**(const unsigned char * key, const unsigned char * in, word32 inSz, unsigned char * out, unsigned char outSz)This function one-shots the data using SipHash to calculate a MAC based on the key.

B.52.2 Functions Documentation

```
int wc_InitSipHash(
    SipHash * siphash,
    const unsigned char * key,
    unsigned char outSz
)
```

This function initializes SipHash with a key for a MAC size.

Parameters:

- **siphash** pointer to the SipHash structure to use for MACing
- **key** pointer to the 16-byte array
- **outSz** number of bytes to output as MAC

See:

- wc_SipHash
- wc_SipHashUpdate
- wc_SipHashFinal

Return:

- 0 Returned upon successfully initializing
- BAD_FUNC_ARG Returned when siphash or key is NULL
- BAD_FUNC_ARG Returned when outSz is neither 8 nor 16

Example

```
SipHash siphash[1];
unsigned char key[16] = { ... };
byte macSz = 8; // 8 or 16

if ((ret = wc_InitSipHash(siphash, key, macSz)) != 0) {
    WOLFSSL_MSG("wc_InitSipHash failed");
}
else if ((ret = wc_SipHashUpdate(siphash, data, len)) != 0) {
    WOLFSSL_MSG("wc_SipHashUpdate failed");
}
else if ((ret = wc_SipHashFinal(siphash, mac, macSz)) != 0) {
    WOLFSSL_MSG("wc_SipHashFinal failed");
}
```

```
int wc_SipHashUpdate(
    SipHash * siphash,
    const unsigned char * in,
    word32 inSz
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **siphash** pointer to the SipHash structure to use for MACing
- **in** the data to be MACed
- **inSz** size of data to be MACed

See:

- wc_SipHash
- wc_InitSipHash
- wc_SipHashFinal

Return:

- 0 Returned upon successfully adding the data to the MAC
- BAD_FUNC_ARG Returned when siphash is NULL
- BAD_FUNC_ARG Returned when in is NULL and inSz is not zero

Example

```
SipHash siphash[1];
byte data[] = { Data to be MACed };
word32 len = sizeof(data);

if ((ret = wc_InitSipHash(siphash, key, macSz)) != 0) {
```

```
    WOLFSSL_MSG("wc_InitSipHash failed");
}
else if ((ret = wc_SipHashUpdate(siphash, data, len)) != 0) {
    WOLFSSL_MSG("wc_SipHashUpdate failed");
}
else if ((ret = wc_SipHashFinal(siphash, mac, macSz)) != 0) {
    WOLFSSL_MSG("wc_SipHashFinal failed");
}
```

```
int wc_SipHashFinal(
    SipHash * siphash,
    unsigned char * out,
    unsigned char outSz
)
```

Finalizes MACing of data. Result is placed into out.

Parameters:

- **siphash** pointer to the SipHash structure to use for MACing
- **out** Byte array to hold MAC value
- **outSz** number of bytes to output as MAC

See:

- wc_SipHash
- wc_InitSipHash
- wc_SipHashUpdate

Return:

- 0 Returned upon successfully finalizing.
- BAD_FUNC_ARG Returned when siphash of out is NULL
- BAD_FUNC_ARG Returned when outSz is not the same as the initialized value

Example

```
SipHash siphash[1];
byte mac[8] = { ... }; // 8 or 16 bytes
byte macSz = sizeof(mac);

if ((ret = wc_InitSipHash(siphash, key, macSz)) != 0) {
    WOLFSSL_MSG("wc_InitSipHash failed");
}
else if ((ret = wc_SipHashUpdate(siphash, data, len)) != 0) {
    WOLFSSL_MSG("wc_SipHashUpdate failed");
}
else if ((ret = wc_SipHashFinal(siphash, mac, macSz)) != 0) {
    WOLFSSL_MSG("wc_SipHashFinal failed");
}
```

```
int wc_SipHash(
    const unsigned char * key,
    const unsigned char * in,
    word32 inSz,
    unsigned char * out,
    unsigned char outSz
)
```

This function one-shots the data using SipHash to calculate a MAC based on the key.

Parameters:

- **key** pointer to the 16-byte array
- **in** the data to be MACed
- **inSz** size of data to be MACed
- **out** Byte array to hold MAC value
- **outSz** number of bytes to output as MAC

See:

- wc_InitSipHash
- wc_SipHashUpdate
- wc_SipHashFinal

Return:

- 0 Returned upon successfully MACing
- BAD_FUNC_ARG Returned when key or out is NULL
- BAD_FUNC_ARG Returned when in is NULL and inSz is not zero
- BAD_FUNC_ARG Returned when outSz is neither 8 nor 16

Example

```
unsigned char key[16] = { ... };
byte data[] = { Data to be MACed };
word32 len = sizeof(data);
byte mac[8] = { ... }; // 8 or 16 bytes
byte macSz = sizeof(mac);

if ((ret = wc_SipHash(key, data, len, mac, macSz)) != 0) {
    WOLFSSL_MSG("wc_SipHash failed");
}
```

B.53 Algorithms - SLH-DSA (FIPS 205)

B.52.2.4 function wc_SipHash More...

B.53.1 Functions

	Name
int	**wc_SlhDsaKey_Init**(SlhDsaKey * key, enum SlhDsaParam param, void * heap, int devId)Initializes an SLH_DSA key object with the specified parameter set. Must be called before any other SLH_DSA operation. Use wc_SlhDsaKey_Free() to release resources when done.
int	**wc_SlhDsaKey_Init_id**(SlhDsaKey * key, enum SlhDsaParam param, const unsigned char * id, int len, void * heap, int devId)Initializes an SLH_DSA key with a device_side key identifier (id). Equivalent to wc_SlhDsaKey_Init() but also stashes a binary id that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.
int	**wc_SlhDsaKey_Init_label**(SlhDsaKey * key, enum SlhDsaParam param, const char * label, void * heap, int devId)Initializes an SLH_DSA key with a device_side key label. Equivalent to wc_SlhDsaKey_Init() but also stashes a label string that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.
void	**wc_SlhDsaKey_Free**(SlhDsaKey * key)Frees resources associated with an SLH-DSA key object.
int	**wc_SlhDsaKey_MakeKey**(SlhDsaKey * key, WC_RNG * rng)Generates a new SLH_DSA key pair using the RNG for randomness. The key must have been initialized with wc_SlhDsaKey_Init() first.
int	**wc_SlhDsaKey_MakeKeyWithRandom**(SlhDsaKey * key, const byte * sk_seed, word32 sk_seed_len, const byte * sk_prf, word32 sk_prf_len, const byte * pk_seed, word32 pk_seed_len)Generates an SLH-DSA key pair from caller-provided seed material. This is the deterministic key generation interface — given the same seeds, the same key pair is produced.
int	**wc_SlhDsaKey_SignDeterministic**(SlhDsaKey * key, const byte * ctx, byte ctxSz, const byte * msg, word32 msgSz, byte * sig, word32 * sigSz)Signs a message using the SLH_DSA external (pure) interface with deterministic randomness. This is FIPS 205 Algorithm 22 with opt_rand set to PK.seed. The message M is wrapped internally as M' = 0x00

	Name
int	**wc_SlhDsaKey_SignWithRandom**(SlhDsaKey * key, const byte * ctx, byte ctxSz, const byte * msg, word32 msgSz, byte * sig, word32 * sigSz, const byte * addRnd)Signs a message using the SLH_DSA external (pure) interface with caller-provided additional randomness. This is FIPS 205 Algorithm 22 with an explicit opt_rand value.
int	**wc_SlhDsaKey_Sign**(SlhDsaKey * key, const byte * ctx, byte ctxSz, const byte * msg, word32 msgSz, byte * sig, word32 * sigSz, WC_RNG * rng)Signs a message using the SLH_DSA external (pure) interface with RNG-provided randomness. This is the general-purpose signing function that uses the WC_RNG for opt_rand.
int	**wc_SlhDsaKey_Verify**(SlhDsaKey * key, const byte * ctx, byte ctxSz, const byte * msg, word32 msgSz, const byte * sig, word32 sigSz)Verifies an SLH_DSA signature over a message using the external (pure) interface. This is FIPS 205 Algorithm 24. The message is wrapped internally as M' = 0x00
int	**wc_SlhDsaKey_SignMsgDeterministic**(SlhDsaKey * key, const byte * mprime, word32 mprimeSz, byte * sig, word32 * sigSz)Signs using the SLH_DSA internal interface with deterministic randomness. Unlike the external interface, M' is provided directly by the caller — no wrapping is performed. This corresponds to FIPS 205 Algorithm 19 (slh_sign_internal) with opt_rand set to PK.seed.
int	**wc_SlhDsaKey_SignMsgWithRandom**(SlhDsaKey * key, const byte * mprime, word32 mprimeSz, byte * sig, word32 * sigSz, const byte * addRnd)Signs using the SLH_DSA internal interface with caller_provided additional randomness. M' is provided directly — no wrapping is performed. This corresponds to FIPS 205 Algorithm 19 (slh_sign_internal) with an explicit opt_rand value. See wc_SlhDsaKey_SignMsgDeterministic for the M' layout used by HashSLH-DSA.
int	**wc_SlhDsaKey_VerifyMsg**(SlhDsaKey * key, const byte * mprime, word32 mprimeSz, const byte * sig, word32 sigSz)Verifies an SLH_DSA signature using the internal interface. M' is provided directly — no wrapping is performed. This corresponds to FIPS 205 Algorithm 20 (slh_verify_internal).

	Name
int	**wc_SlhDsaKey_SignHashDeterministic**(SlhDsaKey * key, const byte * ctx, byte ctxSz, const byte * hash, word32 hashSz, enum wc_HashType hashType, byte * sig, word32 * sigSz)Signs a caller_pre_hashed message digest using the SLH_DSA external (HashSLH_DSA) interface with deterministic randomness, per FIPS 205 Algorithm 23 with the pre_hash domain separator (0x01). The caller must hash the application message with hashType first and pass the digest as hash; this function does NOT hash its input.
int	**wc_SlhDsaKey_SignHashWithRandom**(SlhDsaKey * key, const byte * ctx, byte ctxSz, const byte * hash, word32 hashSz, enum wc_HashType hashType, byte * sig, word32 * sigSz, const byte * addRnd)Signs a caller_pre_hashed message digest using the SLH_DSA external (HashSLH_DSA) interface with caller-provided additional randomness. The caller must hash the application message with hashType first and pass the digest as hash; this function does NOT hash its input.
int	**wc_SlhDsaKey_SignHash**(SlhDsaKey * key, const byte * ctx, byte ctxSz, const byte * hash, word32 hashSz, enum wc_HashType hashType, byte * sig, word32 * sigSz, WC_RNG * rng)Signs a caller_pre_hashed message digest using the SLH_DSA external (HashSLH_DSA) interface with RNG-provided randomness. The caller must hash the application message with hashType first and pass the digest as hash; this function does NOT hash its input.
int	**wc_SlhDsaKey_VerifyHash**(SlhDsaKey * key, const byte * ctx, byte ctxSz, const byte * hash, word32 hashSz, enum wc_HashType hashType, const byte * sig, word32 sigSz)Verifies an SLH_DSA signature using the external HashSLH_DSA interface (FIPS 205 Algorithm 25). The caller must hash the application message with hashType first and pass the digest as hash; this function does NOT hash its input.
int	**wc_SlhDsaKey_ImportPrivate**(SlhDsaKey * key, const byte * in, word32 inLen)Imports an SLH_DSA private key from a raw byte buffer. The buffer must contain the full private key (4*n bytes: SK.seed
int	**wc_SlhDsaKey_ImportPublic**(SlhDsaKey * key, const byte * in, word32 inLen)Imports an SLH_DSA public key from a raw byte buffer. The buffer must contain PK.seed

	Name
int	**wc_SlhDsaKey_CheckKey**(SlhDsaKey * key)Checks the consistency of an SLH-DSA key. For a key with both private and public components, verifies that the public key matches the private key.
int	**wc_SlhDsaKey_ExportPrivate**(SlhDsaKey * key, byte * out, word32 * outLen)Exports the private key from an SLH_DSA key object into a raw byte buffer (4*n bytes).
int	**wc_SlhDsaKey_ExportPublic**(SlhDsaKey * key, byte * out, word32 * outLen)Exports the public key from an SLH_DSA key object into a raw byte buffer (2*n bytes: PK.seed
int	**wc_SlhDsaKey_PrivateSize**(SlhDsaKey * key)Returns the private key size in bytes for the key's parameter set.
int	**wc_SlhDsaKey_PublicSize**(SlhDsaKey * key)Returns the public key size in bytes for the key's parameter set.
int	**wc_SlhDsaKey_SigSize**(SlhDsaKey * key)Returns the signature size in bytes for the key's parameter set.
int	**wc_SlhDsaKey_PrivateSizeFromParam**(enum SlhDsaParam param)Returns the private key size in bytes for the given parameter set without needing an initialized key object.
int	**wc_SlhDsaKey_PublicSizeFromParam**(enum SlhDsaParam param)Returns the public key size in bytes for the given parameter set without needing an initialized key object.
int	**wc_SlhDsaKey_SigSizeFromParam**(enum SlhDsaParam param)Returns the signature size in bytes for the given parameter set without needing an initialized key object.
int	**wc_SlhDsaKey_PrivateKeyDecode**(const byte * input, word32 * inOutIdx, SlhDsaKey * key, word32 inSz)Decodes a DER_encoded SLH_DSA private key in the PKCS#8 OneAsymmetricKey format defined by RFC 9909. The privateKey OCTET STRING contains the raw concatenation SK.seed
int	**wc_SlhDsaKey_PublicKeyDecode**(const byte * input, word32 * inOutIdx, SlhDsaKey * key, word32 inSz)Decodes a DER_encoded SLH_DSA public key in the SubjectPublicKeyInfo (SPKI) format. The SLH-DSA parameter set is detected from the AlgorithmIdentifier OID and key->params is updated accordingly.

	Name
int	**wc_SlhDsaKey_KeyToDer**(SlhDsaKey * key, byte * output, word32 inLen)Encodes an SLH_DSA private key to DER in the PKCS#8 OneAsymmetricKey format defined by RFC 9909. The privateKey OCTET STRING contains the raw 4*n bytes (SK.seed
int	**wc_SlhDsaKey_PrivateKeyToDer**(SlhDsaKey * key, byte * output, word32 inLen)Encodes an SLH-DSA private key to DER. RFC 9909 packs SK.seed
int	**wc_SlhDsaKey_PublicKeyToDer**(SlhDsaKey * key, byte * output, word32 inLen, int withAlg)Encodes an SLH_DSA public key to DER. When withAlg is non_zero the output is a full SubjectPublicKeyInfo structure (AlgorithmIdentifier plus BIT STRING). When withAlg is zero the output contains the raw public key bytes without the SPKI wrapping.

B.53.2 Detailed Description

SLH-DSA (Stateless Hash-based Digital Signature Algorithm) is a quantum-resistant signature scheme standardized by NIST as FIPS 205. It descends from the SPHINCS+ submission and is stateless: signing does not mutate the private key, so there is no key-state synchronization burden on the application.

Twelve parameter sets are supported, formed by combining a hash family (SHAKE or SHA2), a security category (128/192/256) and a speed/size tradeoff (s = small signatures, f = fast signing). The parameter set is selected when the key is initialized via wc_SlhDsaKey_Init().

B.53.3 Functions Documentation

```
int wc_SlhDsaKey_Init(
    SlhDsaKey * key,
    enum SlhDsaParam param,
    void * heap,
    int devId
)
```

Initializes an SLH-DSA key object with the specified parameter set. Must be called before any other SLH-DSA operation. Use wc_SlhDsaKey_Free() to release resources when done.

Parameters:

- **key** Pointer to the SlhDsaKey to initialize.
- **param** Parameter set to use. One of: SLHDSA_SHAKE128S, SLHDSA_SHAKE128F, SLHDSA_SHAKE192S, SLHDSA_SHAKE192F, SLHDSA_SHAKE256S, SLHDSA_SHAKE256F, SLHDSA_SHA2_128S, SLHDSA_SHA2_128F, SLHDSA_SHA2_192S, SLHDSA_SHA2_192F, SLHDSA_SHA2_256S, SLHDSA_SHA2_256F.
- **heap** Pointer to heap hint for dynamic memory allocation. May be NULL.
- **devId** Device identifier for hardware crypto callbacks. Use INVALID_DEVID for software-only.

See:

- wc_SlhDsaKey_Free
- wc_SlhDsaKey_MakeKey

Return:

- 0 on success.
- BAD_FUNC_ARG if key is NULL or param is invalid.

SLH-DSA (FIPS 205) is a stateless hash-based digital signature algorithm. Parameter sets control the hash function (SHAKE or SHA2), security level (128, 192, 256), and speed/size tradeoff (s = small signatures, f = fast signing).

Example

```
SlhDsaKey key;
int ret;

ret = wc_SlhDsaKey_Init(&key, SLHDSA_SHAKE128F, NULL, INVALID_DEVID);
if (ret != 0) {
    // error initializing key
}
// ... use key ...
wc_SlhDsaKey_Free(&key);
```

```
int wc_SlhDsaKey_Init_id(
    SlhDsaKey * key,
    enum SlhDsaParam param,
    const unsigned char * id,
    int len,
    void * heap,
    int devId
)
```

Initializes an SLH-DSA key with a device-side key identifier (id). Equivalent to wc_SlhDsaKey_Init() but also stashes a binary id that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.

Parameters:

- **key** Pointer to the SlhDsaKey to initialize.
- **param** Parameter set to use (see wc_SlhDsaKey_Init).
- **id** Pointer to the device-side key identifier bytes. May be NULL if len is 0.
- **len** Number of bytes in id. Must be in [0, SLHDSA_MAX_ID_LEN].
- **heap** Pointer to heap hint for dynamic memory allocation. May be NULL.
- **devId** Device identifier for the crypto callback. Should be a registered cb devId, not INVALID_DEVID, for the id to be meaningful.

See:

- wc_SlhDsaKey_Init
- wc_SlhDsaKey_Init_label
- wc_SlhDsaKey_Free

Return:

- 0 on success.
- BAD_FUNC_ARG if key is NULL, or if id is NULL while len > 0.
- BUFFER_E if len is negative or greater than SLHDSA_MAX_ID_LEN.

The id is copied into the key object; the caller may free its buffer immediately after this call returns.

Example

```
SlhDsaKey key;
unsigned char id[8] = { 0x01, 0x02, 0x03, 0x04,
                        0x05, 0x06, 0x07, 0x08 };
int ret;

ret = wc_SlhDsaKey_Init_id(&key, SLHDSA_SHAKE128F, id, sizeof(id),
    NULL, devId);
if (ret != 0) {
    // error initializing key with id
}
// ... use key, the cb resolves id -> device key ...
wc_SlhDsaKey_Free(&key);
```

```
int wc_SlhDsaKey_Init_label(
    SlhDsaKey * key,
    enum SlhDsaParam param,
    const char * label,
    void * heap,
    int devId
)
```

Initializes an SLH-DSA key with a device-side key label. Equivalent to wc_SlhDsaKey_Init() but also stashes a label string that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.

Parameters:

- **key** Pointer to the SlhDsaKey to initialize.
- **param** Parameter set to use (see wc_SlhDsaKey_Init).
- **label** NUL-terminated device-side key label string.
- **heap** Pointer to heap hint for dynamic memory allocation. May be NULL.
- **devId** Device identifier for the crypto callback. Should be a registered cb devId, not INVALID_DEVID, for the label to be meaningful.

See:

- wc_SlhDsaKey_Init

- wc_SlhDsaKey_Init_id
- wc_SlhDsaKey_Free

Return:

- 0 on success.
- BAD_FUNC_ARG if key or label is NULL.
- BUFFER_E if label is empty or longer than SLHDSA_MAX_LABEL_LEN.

The label length is taken via XSTRLEN, so embedded NUL bytes terminate the label. The copy stored in key->label is NOT guaranteed to be NUL-terminated (when the input is exactly SLHDSA_MAX_LABEL_LEN bytes the entire array is consumed). Consumers must read at most key->labelLen bytes — do not pass key->label to C-string APIs.

Example

```
SlhDsaKey key;
int ret;

ret = wc_SlhDsaKey_Init_label(&key, SLHDSA_SHAKE128F,
    "device-key-1", NULL, devId);
if (ret != 0) {
    // error initializing key with label
}
// ... use key, the cb resolves label -> device key ...
wc_SlhDsaKey_Free(&key);
```

```
void wc_SlhDsaKey_Free(
    SlhDsaKey * key
)
```

Frees resources associated with an SLH-DSA key object.

Parameters:

- **key** Pointer to the SlhDsaKey to free. May be NULL.

See: wc_SlhDsaKey_Init

Example

```
SlhDsaKey key;
wc_SlhDsaKey_Init(&key, SLHDSA_SHAKE128F, NULL, INVALID_DEVID);
// ... use key ...
wc_SlhDsaKey_Free(&key);
```

```
int wc_SlhDsaKey_MakeKey(
    SlhDsaKey * key,
    WC_RNG * rng
)
```

Generates a new SLH-DSA key pair using the RNG for randomness. The key must have been initialized with wc_SlhDsaKey_Init() first.

Parameters:

- **key** Pointer to an initialized SlhDsaKey.
- **rng** Pointer to an initialized WC_RNG.

See:

- wc_SlhDsaKey_Init
- wc_SlhDsaKey_MakeKeyWithRandom

Return:

- 0 on success.
- BAD_FUNC_ARG if key or rng is NULL, or key is not initialized.

Example

```
SlhDsaKey key;
WC_RNG rng;
int ret;

wc_InitRng(&rng);
wc_SlhDsaKey_Init(&key, SLHDSA_SHAKE128F, NULL, INVALID_DEVID);
ret = wc_SlhDsaKey_MakeKey(&key, &rng);
if (ret != 0) {
    // error generating key
}
```

```
int wc_SlhDsaKey_MakeKeyWithRandom(
    SlhDsaKey * key,
    const byte * sk_seed,
    word32 sk_seed_len,
    const byte * sk_prf,
    word32 sk_prf_len,
    const byte * pk_seed,
    word32 pk_seed_len
)
```

Generates an SLH-DSA key pair from caller-provided seed material. This is the deterministic key generation interface — given the same seeds, the same key pair is produced.

Parameters:

- **key** Pointer to an initialized SlhDsaKey.
- **sk_seed** Secret key seed (n bytes).
- **sk_seed_len** Length of sk_seed.
- **sk_prf** Secret key PRF seed (n bytes).

- **sk_prf_len** Length of sk_prf.
- **pk_seed** Public key seed (n bytes).
- **pk_seed_len** Length of pk_seed.

See: wc_SlhDsaKey_MakeKey

Return:

- 0 on success.
- BAD_FUNC_ARG if key or any seed pointer is NULL, or lengths do not match the parameter set's n value.

Example

```
SlhDsaKey key;
byte sk_seed[16], sk_prf[16], pk_seed[16]; // n=16 for 128-bit params
int ret;

// fill seeds with known values (e.g. from NIST test vectors)
wc_SlhDsaKey_Init(&key, SLHDSA_SHAKE128F, NULL, INVALID_DEVID);
ret = wc_SlhDsaKey_MakeKeyWithRandom(&key,
    sk_seed, sizeof(sk_seed),
    sk_prf, sizeof(sk_prf),
    pk_seed, sizeof(pk_seed));
```

```
int wc_SlhDsaKey_SignDeterministic(
    SlhDsaKey * key,
    const byte * ctx,
    byte ctxSz,
    const byte * msg,
    word32 msgSz,
    byte * sig,
    word32 * sigSz
)
```

Signs a message using the SLH-DSA external (pure) interface with deterministic randomness. This is FIPS 205 Algorithm 22 with opt_rand set to PK.seed. The message M is wrapped internally as M' = 0x00 || len(ctx) || ctx || M before signing.

Parameters:

- **key** Pointer to a private SlhDsaKey.
- **ctx** Context string for domain separation. May be NULL if ctxSz is 0.
- **ctxSz** Length of the context string (0-255).
- **msg** Pointer to the message to sign. May be NULL if msgSz is 0.
- **msgSz** Length of the message.
- **sig** Buffer to receive the signature.
- **sigSz** On input, size of sig buffer. On output, actual signature length.

See:

- wc_SlhDsaKey_SignWithRandom
- wc_SlhDsaKey_Sign
- wc_SlhDsaKey_Verify

Return:

- 0 on success.
- BAD_FUNC_ARG if key, sig, or sigSz is NULL, or msg is NULL but msgSz is greater than 0.
- BAD_LENGTH_E if the output buffer is too small.

Example

```
SlhDsaKey key;
byte sig[WC_SLHDSA_MAX_SIG_LEN];
word32 sigSz = sizeof(sig);
byte msg[] = "Hello World!";
int ret;

// key already generated via wc_SlhDsaKey_MakeKey()
ret = wc_SlhDsaKey_SignDeterministic(&key, NULL, 0,
    msg, sizeof(msg), sig, &sigSz);
```

```
int wc_SlhDsaKey_SignWithRandom(
    SlhDsaKey * key,
    const byte * ctx,
    byte ctxSz,
    const byte * msg,
    word32 msgSz,
    byte * sig,
    word32 * sigSz,
    const byte * addRnd
)
```

Signs a message using the SLH-DSA external (pure) interface with caller-provided additional randomness. This is FIPS 205 Algorithm 22 with an explicit opt_rand value.

Parameters:

- **key** Pointer to a private SlhDsaKey.
- **ctx** Context string. May be NULL if ctxSz is 0.
- **ctxSz** Length of the context string (0-255).
- **msg** Pointer to the message to sign. May be NULL if msgSz is 0.
- **msgSz** Length of the message.
- **sig** Buffer to receive the signature.
- **sigSz** On input, size of sig buffer. On output, actual signature length.
- **addRnd** Additional randomness (n bytes, where n is the parameter set's security parameter).

See:

- wc_SlhDsaKey_SignDeterministic

- wc_SlhDsaKey_Sign

Return:

- 0 on success.
- BAD_FUNC_ARG if key, sig, sigSz, or addRnd is NULL, or msg is NULL but msgSz is greater than 0.

Example

```
SlhDsaKey key;
byte sig[WC_SLHDSA_MAX_SIG_LEN];
word32 sigSz = sizeof(sig);
byte msg[] = "Hello World!";
byte addRnd[16]; // n=16 for 128-bit params
int ret;

wc_RNG_GenerateBlock(&rng, addRnd, sizeof(addRnd));
ret = wc_SlhDsaKey_SignWithRandom(&key, NULL, 0,
    msg, sizeof(msg), sig, &sigSz, addRnd);
```

```
int wc_SlhDsaKey_Sign(
    SlhDsaKey * key,
    const byte * ctx,
    byte ctxSz,
    const byte * msg,
    word32 msgSz,
    byte * sig,
    word32 * sigSz,
    WC_RNG * rng
)
```

Signs a message using the SLH-DSA external (pure) interface with RNG-provided randomness. This is the general-purpose signing function that uses the WC_RNG for opt_rand.

Parameters:

- **key** Pointer to a private SlhDsaKey.
- **ctx** Context string. May be NULL if ctxSz is 0.
- **ctxSz** Length of the context string (0-255).
- **msg** Pointer to the message to sign. May be NULL if msgSz is 0.
- **msgSz** Length of the message.
- **sig** Buffer to receive the signature.
- **sigSz** On input, size of sig buffer. On output, actual signature length.
- **rng** Pointer to an initialized WC_RNG.

See:

- wc_SlhDsaKey_SignDeterministic
- wc_SlhDsaKey_Verify

Return:

- 0 on success.
- BAD_FUNC_ARG if key, sig, sigSz, or rng is NULL, or msg is NULL but msgSz is greater than 0.

Example

```
SlhDsaKey key;
WC_RNG rng;
byte sig[WC_SLHDSA_MAX_SIG_LEN];
word32 sigSz = sizeof(sig);
byte msg[] = "Hello World!";
int ret;

ret = wc_SlhDsaKey_Sign(&key, NULL, 0,
    msg, sizeof(msg), sig, &sigSz, &rng);
```

```
int wc_SlhDsaKey_Verify(
    SlhDsaKey * key,
    const byte * ctx,
    byte ctxSz,
    const byte * msg,
    word32 msgSz,
    const byte * sig,
    word32 sigSz
)
```

Verifies an SLH-DSA signature over a message using the external (pure) interface. This is FIPS 205 Algorithm 24. The message is wrapped internally as M' = 0x00 || len(ctx) || ctx || M before verification.

Parameters:

- **key** Pointer to a public SlhDsaKey.
- **ctx** Context string. May be NULL if ctxSz is 0.
- **ctxSz** Length of the context string (0-255).
- **msg** Pointer to the message to verify. May be NULL if msgSz is 0.
- **msgSz** Length of the message.
- **sig** Pointer to the signature to verify.
- **sigSz** Length of the signature.

See:

- wc_SlhDsaKey_Sign
- wc_SlhDsaKey_SignDeterministic

Return:

- 0 on success (signature valid).

- BAD_FUNC_ARG if key or sig is NULL, or ctx is NULL but ctxSz is greater than 0, or msg is NULL but msgSz is greater than 0.
- BAD_LENGTH_E if sigSz does not match the parameter set's signature length.
- MISSING_KEY if the public key has not been set.
- SIG_VERIFY_E if the signature is invalid.

Example

```
SlhDsaKey key;
byte sig[...]; // previously generated signature
word32 sigSz;
byte msg[] = "Hello World!";
int ret;

ret = wc_SlhDsaKey_Verify(&key, NULL, 0,
    msg, sizeof(msg), sig, sigSz);
if (ret == 0) {
    // signature is valid
}
```

```
int wc_SlhDsaKey_SignMsgDeterministic(
    SlhDsaKey * key,
    const byte * mprime,
    word32 mprimeSz,
    byte * sig,
    word32 * sigSz
)
```

Signs using the SLH-DSA internal interface with deterministic randomness. Unlike the external interface, M' is provided directly by the caller — no wrapping is performed. This corresponds to FIPS 205 Algorithm 19 (slh_sign_internal) with opt_rand set to PK.seed.

Parameters:

- **key** Pointer to a private SlhDsaKey.
- **mprime** Pointer to the pre-constructed M' message.
- **mprimeSz** Length of M'.
- **sig** Buffer to receive the signature.
- **sigSz** On input, size of sig buffer. On output, actual signature length.

See:

- wc_SlhDsaKey_SignMsgWithRandom
- wc_SlhDsaKey_VerifyMsg
- wc_SlhDsaKey_SignDeterministic
- wc_SlhDsaKey_SignHashDeterministic

Return:

- 0 on success.

- BAD_FUNC_ARG if key, mprime, sig, or sigSz is NULL.
- BAD_LENGTH_E if sigSz is less than the parameter set's signature length.
- MISSING_KEY if the private key has not been set.

Use this when the ACVP signatureInterface=internal test framework or a protocol layer has already constructed M'. For HashSLH-DSA the caller builds M' as 0x01 || ctxSz || ctx || OID(hashType) || PHM and passes it in here, where PHM is the hash of the application message under hashType.

Example

```
SlhDsaKey key;
byte sig[WC_SLHDSA_MAX_SIG_LEN];
word32 sigSz = sizeof(sig);
byte mprime[] = { ... }; // pre-constructed M'
int ret;

ret = wc_SlhDsaKey_SignMsgDeterministic(&key,
    mprime, sizeof(mprime), sig, &sigSz);
```

```
int wc_SlhDsaKey_SignMsgWithRandom(
    SlhDsaKey * key,
    const byte * mprime,
    word32 mprimeSz,
    byte * sig,
    word32 * sigSz,
    const byte * addRnd
)
```

Signs using the SLH-DSA internal interface with caller-provided additional randomness. M' is provided directly — no wrapping is performed. This corresponds to FIPS 205 Algorithm 19 (slh_sign_internal) with an explicit opt_rand value. See wc_SlhDsaKey_SignMsgDeterministic for the M' layout used by HashSLH-DSA.

Parameters:

- **key** Pointer to a private SlhDsaKey.
- **mprime** Pointer to the pre-constructed M' message.
- **mprimeSz** Length of M'.
- **sig** Buffer to receive the signature.
- **sigSz** On input, size of sig buffer. On output, actual signature length.
- **addRnd** Additional randomness (n bytes).

See:

- wc_SlhDsaKey_SignMsgDeterministic
- wc_SlhDsaKey_VerifyMsg
- wc_SlhDsaKey_SignHashWithRandom

Return:

- 0 on success.

- BAD_FUNC_ARG if key, mprime, sig, sigSz, or addRnd is NULL.
- BAD_LENGTH_E if sigSz is less than the parameter set's signature length.
- MISSING_KEY if the private key has not been set.

Example

```
SlhDsaKey key;
byte sig[WC_SLHDSA_MAX_SIG_LEN];
word32 sigSz = sizeof(sig);
byte mprime[] = { ... };
byte addRnd[16];
int ret;

wc_RNG_GenerateBlock(&rng, addRnd, sizeof(addRnd));
ret = wc_SlhDsaKey_SignMsgWithRandom(&key,
    mprime, sizeof(mprime), sig, &sigSz, addRnd);
```

```
int wc_SlhDsaKey_VerifyMsg(
    SlhDsaKey * key,
    const byte * mprime,
    word32 mprimeSz,
    const byte * sig,
    word32 sigSz
)
```

Verifies an SLH-DSA signature using the internal interface. M' is provided directly — no wrapping is performed. This corresponds to FIPS 205 Algorithm 20 (slh_verify_internal).

Parameters:

- **key** Pointer to a public SlhDsaKey.
- **mprime** Pointer to the pre-constructed M' message.
- **mprimeSz** Length of M'.
- **sig** Pointer to the signature to verify.
- **sigSz** Length of the signature.

See:

- wc_SlhDsaKey_SignMsgDeterministic
- wc_SlhDsaKey_Verify
- wc_SlhDsaKey_VerifyHash

Return:

- 0 on success (signature valid).
- BAD_FUNC_ARG if key, mprime, or sig is NULL.
- BAD_LENGTH_E if sigSz does not match the parameter set's signature length.
- MISSING_KEY if the public key has not been set.
- SIG_VERIFY_E if the signature is invalid.

Example

```
SlhDsaKey key;
byte sig[...]; // previously generated signature
word32 sigSz;
byte mprime[] = { ... };
int ret;

ret = wc_SlhDsaKey_VerifyMsg(&key,
    mprime, sizeof(mprime), sig, sigSz);
if (ret == 0) {
    // signature is valid
}
```

```
int wc_SlhDsaKey_SignHashDeterministic(
    SlhDsaKey * key,
    const byte * ctx,
    byte ctxSz,
    const byte * hash,
    word32 hashSz,
    enum wc_HashType hashType,
    byte * sig,
    word32 * sigSz
)
```

Signs a caller-pre-hashed message digest using the SLH-DSA external (HashSLH-DSA) interface with deterministic randomness, per FIPS 205 Algorithm 23 with the pre-hash domain separator (0x01). The caller must hash the application message with hashType first and pass the digest as hash; this function does NOT hash its input.

Parameters:

- **key** Pointer to a private SlhDsaKey.
- **ctx** Context string. May be NULL if ctxSz is 0.
- **ctxSz** Length of the context string (0-255).
- **hash** Pointer to the pre-hashed message digest. hashSz must equal the digest size for hashType.
- **hashSz** Length of the digest in bytes.
- **hashType** Hash algorithm used for pre-hashing (selects OID). Supported: WC_HASH_TYPE_SHA224, WC_HASH_TYPE_SHA256, WC_HASH_TYPE_SHA384, WC_HASH_TYPE_SHA512, WC_HASH_TYPE_SHA512_224, WC_HASH_TYPE_SHA512_256, WC_HASH_TYPE_SHAKE128, WC_HASH_TYPE_SHAKE256, WC_HASH_TYPE_SHA3_22 WC_HASH_TYPE_SHA3_256, WC_HASH_TYPE_SHA3_384, WC_HASH_TYPE_SHA3_512.
- **sig** Buffer to receive the signature.
- **sigSz** On input, size of sig buffer. On output, actual signature length.

See:

- wc_SlhDsaKey_SignHashWithRandom
- wc_SlhDsaKey_SignHash
- wc_SlhDsaKey_VerifyHash
- wc_SlhDsaKey_SignMsgDeterministic

Return:

- 0 on success.
- BAD_FUNC_ARG if key, hash, sig, or sigSz is NULL.
- BAD_FUNC_ARG if hashType is WC_HASH_TYPE_NONE, or if the pre-hash gives less collision strength than the parameter set's security category (FIPS 205 sec. 10.2).
- BAD_LENGTH_E if hashSz does not equal the digest size for hashType (32 for SHAKE128, 64 for SHAKE256 per FIPS 205 Section 10.2.2).
- NOT_COMPILED_IN if hashType is not supported in this build.
- MISSING_KEY if the private key has not been set.

Example

```
SlhDsaKey key;
byte sig[WC_SLHDSA_MAX_SIG_LEN];
word32 sigSz = sizeof(sig);
byte msg[] = "Hello World!";
byte digest[WC_SHA256_DIGEST_SIZE];
int ret;

wc_Sha256Hash(msg, sizeof(msg), digest);
ret = wc_SlhDsaKey_SignHashDeterministic(&key, NULL, 0,
    digest, sizeof(digest), WC_HASH_TYPE_SHA256, sig, &sigSz);
```

```
int wc_SlhDsaKey_SignHashWithRandom(
    SlhDsaKey * key,
    const byte * ctx,
    byte ctxSz,
    const byte * hash,
    word32 hashSz,
    enum wc_HashType hashType,
    byte * sig,
    word32 * sigSz,
    const byte * addRnd
)
```

Signs a caller-pre-hashed message digest using the SLH-DSA external (HashSLH-DSA) interface with caller-provided additional randomness. The caller must hash the application message with hashType first and pass the digest as hash; this function does NOT hash its input.

Parameters:

- **key** Pointer to a private SlhDsaKey.
- **ctx** Context string. May be NULL if ctxSz is 0.
- **ctxSz** Length of the context string (0-255).
- **hash** Pointer to the pre-hashed message digest. hashSz must equal the digest size for hashType.
- **hashSz** Length of the digest in bytes.
- **hashType** Hash algorithm used for pre-hashing (selects OID).
- **sig** Buffer to receive the signature.
- **sigSz** On input, size of sig buffer. On output, actual signature length.

- **addRnd** Additional randomness (n bytes).

See:

- wc_SlhDsaKey_SignHashDeterministic
- wc_SlhDsaKey_VerifyHash
- wc_SlhDsaKey_SignMsgWithRandom

Return:

- 0 on success.
- BAD_FUNC_ARG if key, hash, sig, sigSz, or addRnd is NULL.
- BAD_FUNC_ARG if hashType is WC_HASH_TYPE_NONE, or if the pre-hash gives less collision strength than the parameter set's security category (FIPS 205 sec. 10.2).
- BAD_LENGTH_E if hashSz does not equal the digest size for hashType (32 for SHAKE128, 64 for SHAKE256 per FIPS 205 Section 10.2.2).
- NOT_COMPILED_IN if hashType is not supported in this build.

```
int wc_SlhDsaKey_SignHash(
    SlhDsaKey * key,
    const byte * ctx,
    byte ctxSz,
    const byte * hash,
    word32 hashSz,
    enum wc_HashType hashType,
    byte * sig,
    word32 * sigSz,
    WC_RNG * rng
)
```

Signs a caller-pre-hashed message digest using the SLH-DSA external (HashSLH-DSA) interface with RNG-provided randomness. The caller must hash the application message with hashType first and pass the digest as hash; this function does NOT hash its input.

Parameters:

- **key** Pointer to a private SlhDsaKey.
- **ctx** Context string. May be NULL if ctxSz is 0.
- **ctxSz** Length of the context string (0-255).
- **hash** Pointer to the pre-hashed message digest. hashSz must equal the digest size for hashType.
- **hashSz** Length of the digest in bytes.
- **hashType** Hash algorithm used for pre-hashing (selects OID).
- **sig** Buffer to receive the signature.
- **sigSz** On input, size of sig buffer. On output, actual signature length.
- **rng** Pointer to an initialized WC_RNG.

See:

- wc_SlhDsaKey_SignHashDeterministic
- wc_SlhDsaKey_VerifyHash

- wc_SlhDsaKey_SignMsgDeterministic

Return:

- 0 on success.
- BAD_FUNC_ARG if key, hash, sig, sigSz, or rng is NULL.
- BAD_FUNC_ARG if hashType is WC_HASH_TYPE_NONE, or if the pre-hash gives less collision strength than the parameter set's security category (FIPS 205 sec. 10.2).
- BAD_LENGTH_E if hashSz does not equal the digest size for hashType (32 for SHAKE128, 64 for SHAKE256 per FIPS 205 Section 10.2.2).
- NOT_COMPILED_IN if hashType is not supported in this build.

```
int wc_SlhDsaKey_VerifyHash(
    SlhDsaKey * key,
    const byte * ctx,
    byte ctxSz,
    const byte * hash,
    word32 hashSz,
    enum wc_HashType hashType,
    const byte * sig,
    word32 sigSz
)
```

Verifies an SLH-DSA signature using the external HashSLH-DSA interface (FIPS 205 Algorithm 25). The caller must hash the application message with hashType first and pass the digest as hash; this function does NOT hash its input.

Parameters:

- **key** Pointer to a public SlhDsaKey.
- **ctx** Context string. May be NULL if ctxSz is 0.
- **ctxSz** Length of the context string (0-255).
- **hash** Pointer to the pre-hashed message digest. hashSz must equal the digest size for hashType.
- **hashSz** Length of the digest in bytes.
- **hashType** Hash algorithm used for pre-hashing (selects OID). Must match the hash used during signing.
- **sig** Pointer to the signature to verify.
- **sigSz** Length of the signature.

See:

- wc_SlhDsaKey_SignHashDeterministic
- wc_SlhDsaKey_Verify
- wc_SlhDsaKey_VerifyMsg

Return:

- 0 on success (signature valid).
- BAD_FUNC_ARG if key, hash, or sig is NULL.

- BAD_FUNC_ARG if hashType is WC_HASH_TYPE_NONE, or if the pre-hash gives less collision strength than the parameter set's security category (FIPS 205 sec. 10.2).
- BAD_LENGTH_E if sigSz does not match the parameter set, or if hashSz does not equal the digest size for hashType (32 for SHAKE128, 64 for SHAKE256 per FIPS 205 Section 10.2.2).
- NOT_COMPILED_IN if hashType is not supported in this build.
- MISSING_KEY if the public key has not been set.
- SIG_VERIFY_E if the signature is invalid.

Example

```
SlhDsaKey key;
byte sig[...];
word32 sigSz;
byte msg[] = "Hello World!";
byte digest[WC_SHA256_DIGEST_SIZE];
int ret;

wc_Sha256Hash(msg, sizeof(msg), digest);
ret = wc_SlhDsaKey_VerifyHash(&key, NULL, 0,
    digest, sizeof(digest), WC_HASH_TYPE_SHA256, sig, sigSz);
if (ret == 0) {
    // signature is valid
}
```

```
int wc_SlhDsaKey_ImportPrivate(
    SlhDsaKey * key,
    const byte * in,
    word32 inLen
)
```

Imports an SLH-DSA private key from a raw byte buffer. The buffer must contain the full private key (4*n bytes: SK.seed || SK.prf || PK.seed || PK.root). After import, the key can be used for signing.

Parameters:

- **key** Pointer to an initialized SlhDsaKey.
- **in** Buffer containing the raw private key bytes.
- **inLen** Length of the input buffer.

See:

- wc_SlhDsaKey_ExportPrivate
- wc_SlhDsaKey_ImportPublic

Return:

- 0 on success.
- BAD_FUNC_ARG if key or in is NULL, or inLen does not match the expected private key size for the parameter set.

Example

```
SlhDsaKey key;
byte privKey[...]; // 4*n bytes
int ret;

wc_SlhDsaKey_Init(&key, SLHDSA_SHAKE128F, NULL, INVALID_DEVID);
ret = wc_SlhDsaKey_ImportPrivate(&key, privKey, sizeof(privKey));
```

```
int wc_SlhDsaKey_ImportPublic(
    SlhDsaKey * key,
    const byte * in,
    word32 inLen
)
```

Imports an SLH-DSA public key from a raw byte buffer. The buffer must contain PK.seed || PK.root (2*n bytes). After import, the key can be used for verification.

Parameters:

- **key** Pointer to an initialized SlhDsaKey.
- **in** Buffer containing the raw public key bytes.
- **inLen** Length of the input buffer.

See:

- wc_SlhDsaKey_ExportPublic
- wc_SlhDsaKey_ImportPrivate

Return:

- 0 on success.
- BAD_FUNC_ARG if key or in is NULL, or inLen does not match the expected public key size.

Example

```
SlhDsaKey key;
byte pubKey[...]; // 2*n bytes
int ret;

wc_SlhDsaKey_Init(&key, SLHDSA_SHAKE128F, NULL, INVALID_DEVID);
ret = wc_SlhDsaKey_ImportPublic(&key, pubKey, sizeof(pubKey));
```

```
int wc_SlhDsaKey_CheckKey(
    SlhDsaKey * key
)
```

Checks the consistency of an SLH-DSA key. For a key with both private and public components, verifies that the public key matches the private key.

Parameters:

- **key** Pointer to the SlhDsaKey to check.

See:

- wc_SlhDsaKey_MakeKey
- wc_SlhDsaKey_ImportPrivate

Return:

- 0 on success (key is valid).
- BAD_FUNC_ARG if key is NULL.

```
int wc_SlhDsaKey_ExportPrivate(
    SlhDsaKey * key,
    byte * out,
    word32 * outLen
)
```

Exports the private key from an SLH-DSA key object into a raw byte buffer (4*n bytes).

Parameters:

- **key** Pointer to the SlhDsaKey containing a private key.
- **out** Buffer to receive the raw private key bytes.
- **outLen** On input, size of out buffer. On output, bytes written.

See:

- wc_SlhDsaKey_ImportPrivate
- wc_SlhDsaKey_ExportPublic

Return:

- 0 on success.
- BAD_FUNC_ARG if key, out, or outLen is NULL.
- MISSING_KEY if the key object holds no private key.
- BUFFER_E if the output buffer is too small.

Example

```
SlhDsaKey key;
byte privKey[4 * 32]; // 4*n for 256-bit params
word32 privKeySz = sizeof(privKey);
int ret;
```

```
ret = wc_SlhDsaKey_ExportPrivate(&key, privKey, &privKeySz);
```

```
int wc_SlhDsaKey_ExportPublic(
    SlhDsaKey * key,
    byte * out,
    word32 * outLen
)
```

Exports the public key from an SLH-DSA key object into a raw byte buffer (2*n bytes: PK.seed || PK.root).

Parameters:

- **key** Pointer to the SlhDsaKey containing a public key.
- **out** Buffer to receive the raw public key bytes.
- **outLen** On input, size of out buffer. On output, bytes written.

See:

- wc_SlhDsaKey_ImportPublic
- wc_SlhDsaKey_ExportPrivate

Return:

- 0 on success.
- BAD_FUNC_ARG if key, out, or outLen is NULL.
- MISSING_KEY if the key object holds no public key.
- BUFFER_E if the output buffer is too small.

Example

```
SlhDsaKey key;
byte pubKey[2 * 32];
word32 pubKeySz = sizeof(pubKey);
int ret;

ret = wc_SlhDsaKey_ExportPublic(&key, pubKey, &pubKeySz);
```

```
int wc_SlhDsaKey_PrivateSize(
    SlhDsaKey * key
)
```

Returns the private key size in bytes for the key's parameter set.

Parameters:

- **key** Pointer to an initialized SlhDsaKey.

See:

- wc_SlhDsaKey_PublicSize
- wc_SlhDsaKey_SigSize
- wc_SlhDsaKey_PrivateSizeFromParam

Return:

- Private key size in bytes (4*n) on success.
- BAD_FUNC_ARG if key is NULL or not initialized.

```
int wc_SlhDsaKey_PublicSize(
    SlhDsaKey * key
)
```

Returns the public key size in bytes for the key's parameter set.

Parameters:

- **key** Pointer to an initialized SlhDsaKey.

See:

- wc_SlhDsaKey_PrivateSize
- wc_SlhDsaKey_SigSize
- wc_SlhDsaKey_PublicSizeFromParam

Return:

- Public key size in bytes (2*n) on success.
- BAD_FUNC_ARG if key is NULL or not initialized.

```
int wc_SlhDsaKey_SigSize(
    SlhDsaKey * key
)
```

Returns the signature size in bytes for the key's parameter set.

Parameters:

- **key** Pointer to an initialized SlhDsaKey.

See:

- wc_SlhDsaKey_PrivateSize
- wc_SlhDsaKey_PublicSize

- wc_SlhDsaKey_SigSizeFromParam

Return:

- Signature size in bytes on success.
- BAD_FUNC_ARG if key is NULL or not initialized.

```
int wc_SlhDsaKey_PrivateSizeFromParam(
    enum SlhDsaParam param
)
```

Returns the private key size in bytes for the given parameter set without needing an initialized key object.

Parameters:

- **param** The SLH-DSA parameter set.

See: wc_SlhDsaKey_PrivateSize

Return:

- Private key size in bytes (4*n) on success.
- BAD_FUNC_ARG if param is invalid.

```
int wc_SlhDsaKey_PublicSizeFromParam(
    enum SlhDsaParam param
)
```

Returns the public key size in bytes for the given parameter set without needing an initialized key object.

Parameters:

- **param** The SLH-DSA parameter set.

See: wc_SlhDsaKey_PublicSize

Return:

- Public key size in bytes (2*n) on success.
- BAD_FUNC_ARG if param is invalid.

```
int wc_SlhDsaKey_SigSizeFromParam(
    enum SlhDsaParam param
)
```

Returns the signature size in bytes for the given parameter set without needing an initialized key object.

Parameters:

- **param** The SLH-DSA parameter set.

See: wc_SlhDsaKey_SigSize

Return:

- Signature size in bytes on success.
- BAD_FUNC_ARG if param is invalid.

```
int wc_SlhDsaKey_PrivateKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    SlhDsaKey * key,
    word32 inSz
)
```

Decodes a DER-encoded SLH-DSA private key in the PKCS#8 OneAsymmetricKey format defined by RFC 9909. The privateKey OCTET STRING contains the raw concatenation SK.seed || SK.prf || PK.seed || PK.root (4*n bytes) directly, without a nested OCTET STRING wrapper as used by Ed25519/Ed448. The SLH-DSA parameter set is detected from the AlgorithmIdentifier OID and key->params is updated to match. Available only when WOLFSSL_SLHDSA_VERIFY_ONLY is not defined.

Parameters:

- **input** DER-encoded key data.
- **inOutIdx** On input, starting offset into input. On output, advanced past the parsed key (unchanged on failure).
- **key** SLH-DSA key. Parameter set is auto-detected from the encoded OID.
- **inSz** Total size of input in bytes.

See:

- wc_SlhDsaKey_KeyToDer
- wc_SlhDsaKey_PublicKeyDecode
- wc_SlhDsaKey_ImportPrivate

Return:

- 0 on success.
- BAD_FUNC_ARG if input, inOutIdx, or key is NULL, or inSz is 0.

- ASN_PARSE_E if the DER cannot be parsed as an SLH-DSA private key (malformed input, wrong key size, or trailing-field violation).
- NOT_COMPILED_IN if the OID names an SLH-DSA variant that is not built into this library.

On a failure that is detected before any write to key->sk (BAD_FUNC_ARG, header/OID parse errors, or wrong privateKey length), the key state is left untouched. On a failure detected after wc_SlhDsaKey_ImportPrivate has populated key->sk (a SHA-2 precompute error, or a trailing-field validation error), key->sk is scrubbed with ForceZero and the WC_SLHDSA_FLAG_PRIVATE/PUBLIC flags are cleared so flags can never claim valid bytes that were zeroed. In both rollback cases, key->params and inOutIdx are restored to their pre-call values.

```
int wc_SlhDsaKey_PublicKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    SlhDsaKey * key,
    word32 inSz
)
```

Decodes a DER-encoded SLH-DSA public key in the SubjectPublicKeyInfo (SPKI) format. The SLH-DSA parameter set is detected from the AlgorithmIdentifier OID and key->params is updated accordingly.

Parameters:

- **input** DER-encoded key data, or a raw 2*n public key when key->params is already set.
- **inOutIdx** On input, starting offset into input. On output, advanced past the parsed key (unchanged on failure).
- **key** SLH-DSA key. Parameter set is auto-detected from the encoded OID, or honored as-is in the raw fast path.
- **inSz** Total size of input in bytes.

See:

- wc_SlhDsaKey_PublicKeyToDer
- wc_SlhDsaKey_PrivateKeyDecode
- wc_SlhDsaKey_ImportPublic

Return:

- 0 on success.
- BAD_FUNC_ARG if input, inOutIdx, or key is NULL, or inSz is 0.
- ASN_PARSE_E if the DER cannot be parsed as an SLH-DSA public key.
- NOT_COMPILED_IN if the OID names an SLH-DSA variant that is not built into this library.

As a fast path, if key->params is already set the function first hands the entire window from inOutIdx to inSz to wc_SlhDsaKey_ImportPublic. ImportPublic's length check is the disambiguator: a window of exactly 2*n bytes is accepted as a raw public key (PK.seed || PK.root) and consumed in full; any other length is rejected and the function falls through to SPKI parsing. SPKI input always carries enough AlgorithmIdentifier/BIT STRING overhead that it never collides with the 2*n raw length, so it falls through cleanly. The caller does not need to pre-trim the window to 2*n.

On a failure detected before any write (BAD_FUNC_ARG or a malformed SPKI), the key state is left untouched. On a failure detected after ImportPublic has populated the public half of key->sk (a SHA-2

precompute error), the public half sk[2*n* .. *4*n] is scrubbed and WC_SLHDSA_FLAG_PUBLIC is cleared from the flags; the private half is left intact in case the caller imported it earlier. key->params and inOutIdx are restored to their pre-call values.

```
int wc_SlhDsaKey_KeyToDer(
    SlhDsaKey * key,
    byte * output,
    word32 inLen
)
```

Encodes an SLH-DSA private key to DER in the PKCS#8 OneAsymmetricKey format defined by RFC 9909. The privateKey OCTET STRING contains the raw 4*n bytes (SK.seed || SK.prf || PK.seed || PK.root) directly, without the nested OCTET STRING wrapping used by Ed25519/Ed448.

Parameters:

- **key** SLH-DSA key with a populated private key.
- **output** Buffer to receive the DER encoding, or NULL to query the required size.
- **inLen** Size of output in bytes (ignored when output is NULL).

See:

- wc_SlhDsaKey_PrivateKeyDecode
- wc_SlhDsaKey_PrivateKeyToDer
- wc_SlhDsaKey_PublicKeyToDer

Return:

- Size of the encoded DER in bytes on success. Pass NULL as output to query the required buffer size without writing.
- BAD_FUNC_ARG if key or key->params is NULL.
- MISSING_KEY if the private key has not been set.
- BUFFER_E if output is non-NULL and inLen is smaller than the required size.
- NOT_COMPILED_IN if key->params names an SLH-DSA variant whose parameter set is not built in.

Available only when WOLFSSL_SLHDSA_VERIFY_ONLY is not defined and WC_ENABLE_ASYM_KEY_EXPORT is set.

```
int wc_SlhDsaKey_PrivateKeyToDer(
    SlhDsaKey * key,
    byte * output,
    word32 inLen
)
```

Encodes an SLH-DSA private key to DER. RFC 9909 packs SK.seed || SK.prf || PK.seed || PK.root into a single OCTET STRING, so SLH-DSA has no distinct private-only encoding. This function is an intentional alias of wc_SlhDsaKey_KeyToDer, kept for API parity with Ed25519/Ed448 which do have a separate private form.

Parameters:

- **key** SLH-DSA key with a populated private key.
- **output** Buffer to receive the DER encoding, or NULL to query the required size.
- **inLen** Size of output in bytes (ignored when output is NULL).

See:

- wc_SlhDsaKey_KeyToDer
- wc_SlhDsaKey_PrivateKeyDecode

Return:

- Size of the encoded DER in bytes on success. Pass NULL as output to query the required buffer size.
- BAD_FUNC_ARG if key or key->params is NULL.
- MISSING_KEY if the private key has not been set.
- BUFFER_E if output is non-NULL and inLen is smaller than the required size.
- NOT_COMPILED_IN if key->params names an SLH-DSA variant whose parameter set is not built in.

Available only when WOLFSSL_SLHDSA_VERIFY_ONLY is not defined and WC_ENABLE_ASYM_KEY_EXPORT is set.

Return codes are inherited unchanged from wc_SlhDsaKey_KeyToDer.

```
int wc_SlhDsaKey_PublicKeyToDer(
    SlhDsaKey * key,
    byte * output,
    word32 inLen,
    int withAlg
)
```

Encodes an SLH-DSA public key to DER. When withAlg is non-zero the output is a full SubjectPublicKeyInfo structure (AlgorithmIdentifier plus BIT STRING). When withAlg is zero the output contains the raw public key bytes without the SPKI wrapping.

Parameters:

- **key** SLH-DSA key with a populated public key.
- **output** Buffer to receive the DER encoding, or NULL to query the required size.
- **inLen** Size of output in bytes (ignored when output is NULL).
- **withAlg** Non-zero to emit SubjectPublicKeyInfo (with AlgorithmIdentifier); zero to emit the raw public key only.

See:

- wc_SlhDsaKey_PublicKeyDecode
- wc_SlhDsaKey_KeyToDer

Return:

- Size of the encoded DER in bytes on success. Pass NULL as output to query the required buffer size.
- BAD_FUNC_ARG if key or key->params is NULL.
- BUFFER_E if output is non-NULL and inLen is smaller than the required size.
- NOT_COMPILED_IN if key->params names an SLH-DSA variant whose parameter set is not built in.

Available only when WC_ENABLE_ASYM_KEY_EXPORT is set.

B.54 Algorithm - SRTP KDF

B.53.3.33 function wc_SlhDsaKey_PublicKeyToDer

B.54.1 Functions

	Name
int	**wc_SRTP_KDF**(const byte * key, word32 keySz, const byte * salt, word32 saltSz, int kdrIdx, const byte * idx, byte * key1, word32 key1Sz, byte * key2, word32 key2Sz, byte * key3, word32 key3Sz)This function derives keys using SRTP KDF algorithm.
int	**wc_SRTCP_KDF**(const byte * key, word32 keySz, const byte * salt, word32 saltSz, int kdrIdx, const byte * idx, byte * key1, word32 key1Sz, byte * key2, word32 key2Sz, byte * key3, word32 key3Sz)This function derives keys using SRTCP KDF algorithm.
int	**wc_SRTP_KDF_label**(const byte * key, word32 keySz, const byte * salt, word32 saltSz, int kdrIdx, const byte * idx, byte label, byte * outKey, word32 outKeySz)This function derives a key with label using SRTP KDF algorithm.
int	**wc_SRTCP_KDF_label**(const byte * key, word32 keySz, const byte * salt, word32 saltSz, int kdrIdx, const byte * idx, byte label, byte * outKey, word32 outKeySz)This function derives key with label using SRTCP KDF algorithm.
int	**wc_SRTP_KDF_kdr_to_idx**(word32 kdr)This function converts a kdr value to an index to use in SRTP/SRTCP KDF API.

B.54.2 Functions Documentation

```
int wc_SRTP_KDF(
    const byte * key,
    word32 keySz,
    const byte * salt,
```

```
    word32 saltSz,
    int kdrIdx,
    const byte * idx,
    byte * key1,
    word32 key1Sz,
    byte * key2,
    word32 key2Sz,
    byte * key3,
    word32 key3Sz
)
```

This function derives keys using SRTP KDF algorithm.

Parameters:

- **key** Key to use with encryption.
- **keySz** Size of key in bytes.
- **salt** Random non-secret value.
- **saltSz** Size of random in bytes.
- **kdrIdx** Key derivation rate. kdr = 0 when -1, otherwise kdr = 2^kdrIdx.
- **idx** Index value to XOR in.
- **key1** First key. Label value of 0x00.
- **key1Sz** Size of first key in bytes.
- **key2** Second key. Label value of 0x01.
- **key2Sz** Size of second key in bytes.
- **key3** Third key. Label value of 0x02.
- **key3Sz** Size of third key in bytes.

See:

- wc_SRTCP_KDF
- wc_SRTP_KDF_label
- wc_SRTCP_KDF_label
- wc_SRTP_KDF_kdr_to_idx

Return:

- 0 Returned upon successful key derivation.
- BAD_FUNC_ARG Returned when key or salt is NULL
- BAD_FUNC_ARG Returned when key length is not 16, 24 or 32.
- BAD_FUNC_ARG Returned when saltSz is larger than 14.
- BAD_FUNC_ARG Returned when kdrIdx is less than -1 or larger than 24.
- MEMORY_E on dynamic memory allocation failure.

Example

```
unsigned char key[16] = { ... };
unsigned char salt[14] = { ... };
unsigned char idx[6] = { ... };
unsigned char keyE[16];
unsigned char keyA[20];
unsigned char keyS[14];
```

```
int kdrIdx = 0; // Use all of index
int ret;

ret = wc_SRTP_KDF(key, sizeof(key), salt, sizeof(salt), kdrIdx, idx,
    keyE, sizeof(keyE), keyA, sizeof(keyA), keyS, sizeof(keyS));
if (ret != 0) {
    WOLFSSL_MSG("wc_SRTP_KDF failed");
}
```

```
int wc_SRTCP_KDF(
    const byte * key,
    word32 keySz,
    const byte * salt,
    word32 saltSz,
    int kdrIdx,
    const byte * idx,
    byte * key1,
    word32 key1Sz,
    byte * key2,
    word32 key2Sz,
    byte * key3,
    word32 key3Sz
)
```

This function derives keys using SRTCP KDF algorithm.

Parameters:

- **key** Key to use with encryption.
- **keySz** Size of key in bytes.
- **salt** Random non-secret value.
- **saltSz** Size of random in bytes.
- **kdrIdx** Key derivation rate. kdr = 0 when -1, otherwise kdr = 2^kdrIdx.
- **idx** Index value to XOR in.
- **key1** First key. Label value of 0x03 (WC_SRTCP_LABEL_ENCRYPTION).
- **key1Sz** Size of first key in bytes.
- **key2** Second key. Label value of 0x04 (WC_SRTCP_LABEL_MSG_AUTH).
- **key2Sz** Size of second key in bytes.
- **key3** Third key. Label value of 0x05 (WC_SRTCP_LABEL_SALT).
- **key3Sz** Size of third key in bytes.

See:

- wc_SRTP_KDF
- wc_SRTP_KDF_label
- wc_SRTCP_KDF_label
- wc_SRTP_KDF_kdr_to_idx

Return:

- 0 Returned upon successful key derivation.

- BAD_FUNC_ARG Returned when key or salt is NULL
- BAD_FUNC_ARG Returned when key length is not 16, 24 or 32.
- BAD_FUNC_ARG Returned when saltSz is larger than 14.
- BAD_FUNC_ARG Returned when kdrIdx is less than -1 or larger than 24.
- MEMORY_E on dynamic memory allocation failure.

Example

```
unsigned char key[16] = { ... };
unsigned char salt[14] = { ... };
unsigned char idx[4] = { ... };
unsigned char keyE[16];
unsigned char keyA[20];
unsigned char keyS[14];
int kdrIdx = 0; // Use all of index
int ret;

ret = wc_SRTCP_KDF(key, sizeof(key), salt, sizeof(salt), kdrIdx, idx,
    keyE, sizeof(keyE), keyA, sizeof(keyA), keyS, sizeof(keyS));
if (ret != 0) {
    WOLFSSL_MSG("wc_SRTP_KDF failed");
}
```

```
int wc_SRTP_KDF_label(
    const byte * key,
    word32 keySz,
    const byte * salt,
    word32 saltSz,
    int kdrIdx,
    const byte * idx,
    byte label,
    byte * outKey,
    word32 outKeySz
)
```

This function derives a key with label using SRTP KDF algorithm.

Parameters:

- **key** Key to use with encryption.
- **keySz** Size of key in bytes.
- **salt** Random non-secret value.
- **saltSz** Size of random in bytes.
- **kdrIdx** Key derivation rate. kdr = 0 when -1, otherwise kdr = 2^kdrIdx.
- **idx** Index value to XOR in.
- **label** Label to use when deriving key.
- **outKey** Derived key.
- **outKeySz** Size of derived key in bytes.

See:

- wc_SRTP_KDF
- wc_SRTCP_KDF
- wc_SRTCP_KDF_label
- wc_SRTP_KDF_kdr_to_idx

Return:

- 0 Returned upon successful key derivation.
- BAD_FUNC_ARG Returned when key, salt or outKey is NULL
- BAD_FUNC_ARG Returned when key length is not 16, 24 or 32.
- BAD_FUNC_ARG Returned when saltSz is larger than 14.
- BAD_FUNC_ARG Returned when kdrIdx is less than -1 or larger than 24.
- MEMORY_E on dynamic memory allocation failure.

Example

```
unsigned char key[16] = { ... };
unsigned char salt[14] = { ... };
unsigned char idx[6] = { ... };
unsigned char keyE[16];
int kdrIdx = 0; // Use all of index
int ret;

ret = wc_SRTP_KDF_label(key, sizeof(key), salt, sizeof(salt), kdrIdx, idx,
    WC_SRTP_LABEL_ENCRYPTION, keyE, sizeof(keyE));
if (ret != 0) {
    WOLFSSL_MSG("wc_SRTP_KDF failed");
}
```

```
int wc_SRTCP_KDF_label(
    const byte * key,
    word32 keySz,
    const byte * salt,
    word32 saltSz,
    int kdrIdx,
    const byte * idx,
    byte label,
    byte * outKey,
    word32 outKeySz
)
```

This function derives key with label using SRTCP KDF algorithm.

Parameters:

- **key** Key to use with encryption.
- **keySz** Size of key in bytes.
- **salt** Random non-secret value.
- **saltSz** Size of random in bytes.
- **kdrIdx** Key derivation rate. kdr = 0 when -1, otherwise kdr = 2^kdrIdx.
- **idx** Index value to XOR in.

- **label** Label to use when deriving key.
- **outKey** Derived key.
- **outKeySz** Size of derived key in bytes.

See:

- wc_SRTP_KDF
- wc_SRTCP_KDF
- wc_SRTP_KDF_label
- wc_SRTP_KDF_kdr_to_idx

Return:

- 0 Returned upon successful key derivation.
- BAD_FUNC_ARG Returned when key, salt or outKey is NULL
- BAD_FUNC_ARG Returned when key length is not 16, 24 or 32.
- BAD_FUNC_ARG Returned when saltSz is larger than 14.
- BAD_FUNC_ARG Returned when kdrIdx is less than -1 or larger than 24.
- MEMORY_E on dynamic memory allocation failure.

Example

```
unsigned char key[16] = { ... };
unsigned char salt[14] = { ... };
unsigned char idx[4] = { ... };
unsigned char keyE[16];
int kdrIdx = 0; // Use all of index
int ret;

ret = wc_SRTCP_KDF_label(key, sizeof(key), salt, sizeof(salt), kdrIdx,
    idx, WC_SRTCP_LABEL_ENCRYPTION, keyE, sizeof(keyE));
if (ret != 0) {
    WOLFSSL_MSG("wc_SRTP_KDF failed");
}
```

```
int wc_SRTP_KDF_kdr_to_idx(
    word32 kdr
)
```

This function converts a kdr value to an index to use in SRTP/SRTCP KDF API.

Parameters:

- **kdr** Key derivation rate to convert.

See:

- wc_SRTP_KDF
- wc_SRTCP_KDF
- wc_SRTP_KDF_label

- wc_SRTCP_KDF_label

Return: Key derivation rate as an index.

Example

```
word32 kdr = 0x00000010;
int kdrIdx;
int ret;

kdrIdx = wc_SRTP_KDF_kdr_to_idx(kdr);
```

B.55 Algorithms - SRP

B.54.2.5 function wc_SRTP_KDF_kdr_to_idx

B.55.1 Functions

	Name
int	**wc_SrpInit**(Srp * srp, SrpType type, SrpSide side)Initializes the Srp struct for usage.
int	**wc_SrpInit_ex**(Srp * srp, SrpType type, SrpSide side, void * heap, int devId)Initializes the Srp struct for usage with extended parameters. This function is similar to wc_SrpInit but allows specification of a custom heap hint and device ID for hardware acceleration.
void	**wc_SrpTerm**(Srp * srp)Releases the Srp struct resources after usage.
int	**wc_SrpSetUsername**(Srp * srp, const byte * username, word32 size)Sets the username. This function MUST be called after wc_SrpInit.
int	**wc_SrpSetParams**(Srp * srp, const byte * N, word32 nSz, const byte * g, word32 gSz, const byte * salt, word32 saltSz)Sets the srp parameters based on the username.. Must be called after wc_SrpSetUsername.
int	**wc_SrpSetPassword**(Srp * srp, const byte * password, word32 size)Sets the password. Setting the password does not persists the clear password data in the srp structure. The client calculates x = H(salt + H(user:pswd)) and stores it in the auth field. This function MUST be called after wc_SrpSetParams and is CLIENT SIDE ONLY.
int	**wc_SrpSetVerifier**(Srp * srp, const byte * verifier, word32 size)Sets the verifier. This function MUST be called after wc_SrpSetParams and is SERVER SIDE ONLY.

	Name
int	**wc_SrpGetVerifier**(Srp * srp, byte * verifier, word32 * size)Gets the verifier. The client calculates the verifier with v = g ^ x % N. This function MAY be called after wc_SrpSetPassword and is CLIENT SIDE ONLY.
int	**wc_SrpSetPrivate**(Srp * srp, const byte * priv, word32 size)Sets the private ephemeral value. The private ephemeral value is known as: a at the client side. a = random() b at the server side. b = random() This function is handy for unit test cases or if the developer wants to use an external random source to set the ephemeral value. This function MAY be called before wc_SrpGetPublic.
int	**wc_SrpGetPublic**(Srp * srp, byte * pub, word32 * size)Gets the public ephemeral value. The public ephemeral value is known as: A at the client side. A = g ^ a % N B at the server side. B = (k * v + (g ^ b % N)) % N This function MUST be called after wc_SrpSetPassword or wc_SrpSetVerifier. The function wc_SrpSetPrivate may be called before wc_SrpGetPublic.
int	**wc_SrpComputeKey**(Srp * srp, byte * clientPubKey, word32 clientPubKeySz, byte * serverPubKey, word32 serverPubKeySz)Computes the session key. The key can be accessed at srp->key after success.
int	**wc_SrpGetProof**(Srp * srp, byte * proof, word32 * size)Gets the proof. This function MUST be called after wc_SrpComputeKey.
int	**wc_SrpVerifyPeersProof**(Srp * srp, byte * proof, word32 size)Verifies the peers proof. This function MUST be called before wc_SrpGetSessionKey.

B.55.2 Functions Documentation

```
int wc_SrpInit(
    Srp * srp,
    SrpType type,
    SrpSide side
)
```

Initializes the Srp struct for usage.

Parameters:

- **srp** the Srp structure to be initialized.
- **type** the hash type to be used.
- **side** the side of the communication.

See:

- wc_SrpTerm
- wc_SrpSetUsername

Return:

- 0 on success.
- BAD_FUNC_ARG Returns when there's an issue with the arguments such as srp being null or SrpSide not being SRP_CLIENT_SIDE or SRP_SERVER_SIDE.
- NOT_COMPILED_IN Returns when a type is passed as an argument but hasn't been configured in the wolfCrypt build.
- <0 on error.

Example

```
Srp srp;
if (wc_SrpInit(&srp, SRP_TYPE_SHA, SRP_CLIENT_SIDE) != 0)
{
    // Initialization error
}
else
{
    wc_SrpTerm(&srp);
}
```

```
int wc_SrpInit_ex(
    Srp * srp,
    SrpType type,
    SrpSide side,
    void * heap,
    int devId
)
```

Initializes the Srp struct for usage with extended parameters. This function is similar to wc_SrpInit but allows specification of a custom heap hint and device ID for hardware acceleration.

Parameters:

- **srp** the Srp structure to be initialized.
- **type** the hash type to be used.
- **side** the side of the communication.
- **heap** pointer to heap hint for memory allocation (can be NULL).
- **devId** device ID for hardware acceleration (use INVALID_DEVID for software only).

See:

- wc_SrpInit
- wc_SrpTerm

- wc_SrpSetUsername

Return:

- 0 on success.
- BAD_FUNC_ARG Returns when there's an issue with the arguments such as srp being null or SrpSide not being SRP_CLIENT_SIDE or SRP_SERVER_SIDE.
- NOT_COMPILED_IN Returns when a type is passed as an argument but hasn't been configured in the wolfCrypt build.
- <0 on error.

Example

```
Srp srp;
void* heap = NULL;
int devId = INVALID_DEVID;

if (wc_SrpInit_ex(&srp, SRP_TYPE_SHA, SRP_CLIENT_SIDE, heap,
                  devId) != 0) {
    // Initialization error
}
else {
    wc_SrpTerm(&srp);
}
```

```
void wc_SrpTerm(
    Srp * srp
)
```

Releases the Srp struct resources after usage.

Parameters:

- **srp** Pointer to the Srp structure to be terminated.

See: wc_SrpInit

Return: none No returns.

Example

```
Srp srp;
wc_SrpInit(&srp, SRP_TYPE_SHA, SRP_CLIENT_SIDE);
// Use srp
wc_SrpTerm(&srp)
```

```
int wc_SrpSetUsername(
    Srp * srp,
    const byte * username,
```

```
    word32 size
)
```

Sets the username. This function MUST be called after wc_SrpInit.

Parameters:

- **srp** the Srp structure.
- **username** the buffer containing the username.
- **size** the username size in bytes

See:

- wc_SrpInit
- wc_SrpSetParams
- wc_SrpTerm

Return:

- 0 Username set successfully.
- BAD_FUNC_ARG: Return if srp or username is null.
- MEMORY_E: Returns if there is an issue allocating memory for srp->user
- < 0: Error.

Example

```
Srp srp;
byte username[] = "user";
word32 usernameSize = 4;

wc_SrpInit(&srp, SRP_TYPE_SHA, SRP_CLIENT_SIDE);
if(wc_SrpSetUsername(&srp, username, usernameSize) != 0)
{
    // Error occurred setting username.
}
wc_SrpTerm(&srp);
```

```
int wc_SrpSetParams(
    Srp * srp,
    const byte * N,
    word32 nSz,
    const byte * g,
    word32 gSz,
    const byte * salt,
    word32 saltSz
)
```

Sets the srp parameters based on the username.. Must be called after wc_SrpSetUsername.

Parameters:

- **srp** the Srp structure.
- **N** the Modulus. N = 2q+1, [q, N] are primes.
- **nSz** the N size in bytes.
- **g** the Generator modulo N.
- **gSz** the g size in bytes
- **salt** a small random salt. Specific for each username.
- **saltSz** the salt size in bytes

See:

- wc_SrpInit
- wc_SrpSetUsername
- wc_SrpTerm

Return:

- 0 Success
- BAD_FUNC_ARG Returns if srp, N, g, or salt is null or if nSz < gSz.
- SRP_CALL_ORDER_E Returns if wc_SrpSetParams is called before wc_SrpSetUsername.
- <0 Error

Example

```
Srp srp;
byte username[] = "user";
word32 usernameSize = 4;

byte N[] = { }; // Contents of byte array N
byte g[] = { }; // Contents of byte array g
byte salt[] = { }; // Contents of byte array salt

wc_SrpInit(&srp, SRP_TYPE_SHA, SRP_CLIENT_SIDE);
wc_SrpSetUsername(&srp, username, usernameSize);

if(wc_SrpSetParams(&srp, N, sizeof(N), g, sizeof(g), salt,
sizeof(salt)) != 0)
{
    // Error setting params
}
wc_SrpTerm(&srp);
```

```
int wc_SrpSetPassword(
    Srp * srp,
    const byte * password,
    word32 size
)
```

Sets the password. Setting the password does not persists the clear password data in the srp structure. The client calculates x = H(salt + H(user:pswd)) and stores it in the auth field. This function MUST be called after wc_SrpSetParams and is CLIENT SIDE ONLY.

Parameters:

- **srp** The Srp structure.
- **password** The buffer containing the password.
- **size** The size of the password in bytes.

See:

- wc_SrpInit
- wc_SrpSetUsername
- wc_SrpSetParams

Return:

- 0 Success
- BAD_FUNC_ARG Returns if srp or password is null or if srp->side is not set to SRP_CLIENT_SIDE.
- SRP_CALL_ORDER_E Returns when wc_SrpSetPassword is called out of order.
- <0 Error

Example

```
Srp srp;
byte username[] = "user";
word32 usernameSize = 4;
byte password[] = "password";
word32 passwordSize = 8;

byte N[] = { }; // Contents of byte array N
byte g[] = { }; // Contents of byte array g
byte salt[] = { }; // Contents of byte array salt

wc_SrpInit(&srp, SRP_TYPE_SHA, SRP_CLIENT_SIDE);
wc_SrpSetUsername(&srp, username, usernameSize);
wc_SrpSetParams(&srp, N, sizeof(N), g, sizeof(g), salt, sizeof(salt));

if(wc_SrpSetPassword(&srp, password, passwordSize) != 0)
{
    // Error setting password
}

wc_SrpTerm(&srp);
```

```
int wc_SrpSetVerifier(
    Srp * srp,
    const byte * verifier,
    word32 size
)
```

Sets the verifier. This function MUST be called after wc_SrpSetParams and is SERVER SIDE ONLY.

Parameters:

- **srp** The Srp structure.
- **verifier** The structure containing the verifier.
- **size** The verifier size in bytes.

See:

- wc_SrpInit
- wc_SrpSetParams
- wc_SrpGetVerifier

Return:

- 0 Success
- BAD_FUNC_ARG Returned if srp or verifier is null or srp->side is not SRP_SERVER_SIDE.
- <0 Error

Example

```
Srp srp;
byte username[] = "user";
word32 usernameSize = 4;

byte N[] = { }; // Contents of byte array N
byte g[] = { }; // Contents of byte array g
byte salt[] = { }; // Contents of byte array salt
wc_SrpInit(&srp, SRP_TYPE_SHA, SRP_SERVER_SIDE);
wc_SrpSetUsername(&srp, username, usernameSize);
wc_SrpSetParams(&srp, N, sizeof(N), g, sizeof(g), salt, sizeof(salt))
byte verifier[] = { }; // Contents of some verifier

if(wc_SrpSetVerifier(&srp, verifier, sizeof(verifier)) != 0)
{
    // Error setting verifier
}

wc_SrpTerm(&srp);
```

```
int wc_SrpGetVerifier(
    Srp * srp,
    byte * verifier,
    word32 * size
)
```

Gets the verifier. The client calculates the verifier with v = g ^ x % N. This function MAY be called after wc_SrpSetPassword and is CLIENT SIDE ONLY.

Parameters:

- **srp** The Srp structure.
- **verifier** The buffer to write the verifier.
- **size** Buffer size in bytes. Updated with the verifier size.

See:

- wc_SrpSetVerifier
- wc_SrpSetPassword

Return:

- 0 Success
- BAD_FUNC_ARG Returned if srp, verifier or size is null or if srp->side is not SRP_CLIENT_SIDE.
- SRP_CALL_ORDER_E Returned if wc_SrpGetVerifier is called out of order.
- <0 Error

Example

```
Srp srp;
byte username[] = "user";
word32 usernameSize = 4;
byte password[] = "password";
word32 passwordSize = 8;

byte N[] = { }; // Contents of byte array N
byte g[] = { }; // Contents of byte array g
byte salt[] = { }; // Contents of byte array salt
byte v[64];
word32 vSz = 0;
vSz = sizeof(v);

wc_SrpInit(&srp, SRP_TYPE_SHA, SRP_CLIENT_SIDE);
wc_SrpSetUsername(&srp, username, usernameSize);
wc_SrpSetParams(&srp, N, sizeof(N), g, sizeof(g), salt, sizeof(salt))
wc_SrpSetPassword(&srp, password, passwordSize)

if( wc_SrpGetVerifier(&srp, v, &vSz ) != 0)
{
    // Error getting verifier
}
wc_SrpTerm(&srp);
```

```
int wc_SrpSetPrivate(
    Srp * srp,
    const byte * priv,
    word32 size
)
```

Sets the private ephemeral value. The private ephemeral value is known as: a at the client side. a = random() b at the server side. b = random() This function is handy for unit test cases or if the developer

wants to use an external random source to set the ephemeral value. This function MAY be called before wc_SrpGetPublic.

Parameters:

- **srp** the Srp structure.
- **priv** the ephemeral value.
- **size** the private size in bytes.

See: wc_SrpGetPublic

Return:

- 0 Success
- BAD_FUNC_ARG Returned if srp, private, or size is null.
- SRP_CALL_ORDER_E Returned if wc_SrpSetPrivate is called out of order.
- <0 Error

Example

```
Srp srp;
byte username[] = "user";
word32 usernameSize = 4;

byte N[] = { }; // Contents of byte array N
byte g[] = { }; // Contents of byte array g
byte salt[] = { }; // Contents of byte array salt
byte verifier = { }; // Contents of some verifier
wc_SrpInit(&srp, SRP_TYPE_SHA, SRP_SERVER_SIDE);
wc_SrpSetUsername(&srp, username, usernameSize);
wc_SrpSetParams(&srp, N, sizeof(N), g, sizeof(g), salt, sizeof(salt))
wc_SrpSetVerifier(&srp, verifier, sizeof(verifier))

byte b[] = { }; // Some ephemeral value
if( wc_SrpSetPrivate(&srp, b, sizeof(b)) != 0)
{
    // Error setting private ephemeral
}

wc_SrpTerm(&srp);
```

```
int wc_SrpGetPublic(
    Srp * srp,
    byte * pub,
    word32 * size
)
```

Gets the public ephemeral value. The public ephemeral value is known as: A at the client side. A = g ^ a % N B at the server side. B = (k * v + (g ^ b % N)) % N This function MUST be called after wc_SrpSetPassword or wc_SrpSetVerifier. The function wc_SrpSetPrivate may be called before wc_SrpGetPublic.

Parameters:

- **srp** the Srp structure.
- **pub** the buffer to write the public ephemeral value.
- **size** IN: the buffer size in bytes. OUT: Will be updated with the ephemeral value size.

See:

- wc_SrpSetPrivate
- wc_SrpSetPassword
- wc_SrpSetVerifier

Return:

- 0 Success
- BAD_FUNC_ARG Returned if srp, pub, or size is null.
- SRP_CALL_ORDER_E Returned if wc_SrpGetPublic is called out of order.
- BUFFER_E Returned if size < srp.N.
- <0 Error

Caller must observe value of size upon return to know the actual size.

Example

```
Srp srp;
byte username[] = "user";
word32 usernameSize = 4;
byte password[] = "password";
word32 passwordSize = 8;

byte N[] = { }; // Contents of byte array N
byte g[] = { }; // Contents of byte array g
byte salt[] = { }; // Contents of byte array salt
wc_SrpInit(&srp, SRP_TYPE_SHA, SRP_CLIENT_SIDE);
wc_SrpSetUsername(&srp, username, usernameSize);
wc_SrpSetParams(&srp, N, sizeof(N), g, sizeof(g), salt, sizeof(salt));
wc_SrpSetPassword(&srp, password, passwordSize)

byte public[64];
word32 publicSz = sizeof(public);

if( wc_SrpGetPublic(&srp, public, &publicSz) != 0)
{
    // Error getting public ephemeral
}

wc_SrpTerm(&srp);
```

```
int wc_SrpComputeKey(
    Srp * srp,
```

```
    byte * clientPubKey,
    word32 clientPubKeySz,
    byte * serverPubKey,
    word32 serverPubKeySz
)
```

Computes the session key. The key can be accessed at srp->key after success.

Parameters:

- **srp** the Srp structure.
- **clientPubKey** the client's public ephemeral value.
- **clientPubKeySz** the client's public ephemeral value size.
- **serverPubKey** the server's public ephemeral value.
- **serverPubKeySz** the server's public ephemeral value size.

See: wc_SrpGetPublic

Return:

- 0 Success
- BAD_FUNC_ARG Returned if srp, clientPubKey, or serverPubKey or if clientPubKeySz or server-PubKeySz is 0.
- SRP_CALL_ORDER_E Returned if wc_SrpComputeKey is called out of order.
- <0 Error

Example

```
Srp server;

byte username[] = "user";
    word32 usernameSize = 4;
byte password[] = "password";
word32 passwordSize = 8;
byte N[] = { }; // Contents of byte array N
byte g[] = { }; // Contents of byte array g
byte salt[] = { }; // Contents of byte array salt
byte verifier[] = { }; // Contents of some verifier
byte serverPubKey[] = { }; // Contents of server pub key
word32 serverPubKeySize = sizeof(serverPubKey);
byte clientPubKey[64];
word32 clientPubKeySize = 64;

wc_SrpInit(&server, SRP_TYPE_SHA, SRP_SERVER_SIDE);
wc_SrpSetUsername(&server, username, usernameSize);
wc_SrpSetParams(&server, N, sizeof(N), g, sizeof(g), salt, sizeof(salt));
wc_SrpSetVerifier(&server, verifier, sizeof(verifier));
wc_SrpGetPublic(&server, serverPubKey, &serverPubKeySize);

wc_SrpComputeKey(&server, clientPubKey, clientPubKeySz,
                                    serverPubKey, serverPubKeySize)
wc_SrpTerm(&server);
```

```
int wc_SrpGetProof(
    Srp * srp,
    byte * proof,
    word32 * size
)
```

Gets the proof. This function MUST be called after wc_SrpComputeKey.

Parameters:

- **srp** the Srp structure.
- **proof** the peers proof.
- **size** the proof size in bytes.

See: wc_SrpComputeKey

Return:

- 0 Success
- BAD_FUNC_ARG Returns if srp, proof, or size is null.
- BUFFER_E Returns if size is less than the hash size of srp->type.
- <0 Error

Example

```
Srp cli;
byte clientProof[SRP_MAX_DIGEST_SIZE];
word32 clientProofSz = SRP_MAX_DIGEST_SIZE;

// Initialize Srp following steps from previous examples

if (wc_SrpGetProof(&cli, clientProof, &clientProofSz) != 0)
{
    // Error getting proof
}
```

```
int wc_SrpVerifyPeersProof(
    Srp * srp,
    byte * proof,
    word32 size
)
```

Verifies the peers proof. This function MUST be called before wc_SrpGetSessionKey.

Parameters:

- **srp** the Srp structure.
- **proof** the peers proof.
- **size** the proof size in bytes.

See:

- wc_SrpGetSessionKey
- wc_SrpGetProof
- wc_SrpTerm

Return:

- 0 Success
- <0 Error

Example

```
Srp cli;
Srp srv;
byte clientProof[SRP_MAX_DIGEST_SIZE];
word32 clientProofSz = SRP_MAX_DIGEST_SIZE;

// Initialize Srp following steps from previous examples
// First get the proof
wc_SrpGetProof(&cli, clientProof, &clientProofSz)

if (wc_SrpVerifyPeersProof(&srv, clientProof, clientProofSz) != 0)
{
    // Error verifying proof
}
```

B.56 Algorithms - XMSS / XMSS^MT (RFC 8391)

B.55.2.13 function wc_SrpVerifyPeersProof More...

B.56.1 Functions

	Name
int	**wc_XmssKey_Init**(XmssKey * key, void * heap, int devId)Initializes an XmssKey object. Must be called before any other XMSS/XMSS^MT operation. Use wc_XmssKey_Free() to release resources when done.
int	**wc_XmssKey_InitId**(XmssKey * key, const unsigned char * id, int len, void * heap, int devId)Initializes an XmssKey with a device_side key identifier. Equivalent to wc_XmssKey_Init() but also stashes a binary id that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.

	Name
int	**wc_XmssKey_InitLabel**(XmssKey * key, const char * label, void * heap, int devId)Initializes an XmssKey with a device_side key label. Equivalent to wc_XmssKey_Init() but also stashes a label string that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.
int	**wc_XmssKey_SetParamStr**(XmssKey * key, const char * str)Selects an XMSS or XMSS^MT parameter set by its RFC 8391 name. Accepted names take the form “XMSS_” (single_tree) or “XMSSMT_/” (multi_tree), for example “XMSS-SHA2_10_256” or “XMSSMT-SHA2_20/2_256”. The set of names actually accepted depends on the hash families and tree heights enabled at build time.
int	**wc_XmssKey_GetParamStr**(const XmssKey * key, const char ** str)Retrieves the parameter set name currently selected on this key. The returned pointer is a static string and must not be freed by the caller.
int	**wc_XmssKey_SetWriteCb**(XmssKey * key, wc_xmss_write_private_key_cb write_cb)Registers the callback that wolfSSL invokes to persist updated private key state. Because XMSS/XMSS^MT is stateful, the application MUST persist the private key after each successful sign before the signature is released; otherwise a crash or restart can lead to one-time key reuse and break the scheme.
int	**wc_XmssKey_SetReadCb**(XmssKey * key, wc_xmss_read_private_key_cb read_cb)Registers the callback that wolfSSL invokes to load persisted private key state. Used by wc_XmssKey_Reload() to bring a saved key back into memory for further signing.
int	**wc_XmssKey_SetContext**(XmssKey * key, void * context)Sets the opaque context pointer passed to both the read and write private-key callbacks.
int	**wc_XmssKey_MakeKey**(XmssKey * key, WC_RNG * rng)Generates a fresh XMSS/XMSS^MT key pair. The parameter set must already be selected via wc_XmssKey_SetParamStr() and the read/write callbacks must be registered. The newly generated private key is persisted via the write callback before the function returns; on success the key transitions to state WC_XMSS_STATE_OK.

	Name
int	**wc_XmssKey_Reload**(XmssKey * key)Reloads a previously generated XMSS/XMSS^MT private key from persistent storage using the registered read callback, restoring the key to a state where it can sign further messages. On success the key is in state WC_XMSS_STATE_OK.
int	**wc_XmssKey_GetPrivLen**(const XmssKey * key, word32 * len)Returns the size in bytes of the encoded private key for the parameter set selected on this key.
int	**wc_XmssKey_GetPubLen**(const XmssKey * key, word32 * len)Returns the size in bytes of the XMSS/XMSS^MT public key for the parameter set selected on this key.
int	**wc_XmssKey_GetSigLen**(const XmssKey * key, word32 * len)Returns the signature size in bytes for the parameter set selected on this key.
int	**wc_XmssKey_Sign**(XmssKey * key, byte * sig, word32 * sigSz, const byte * msg, int msgSz)Signs msg with the XMSS/XMSS^MT private key in key. On entry *sigSz is the size of the sig buffer; on success it is updated to the bytes written.
int	**wc_XmssKey_SigsLeft**(XmssKey * key)Reports whether this key can create another signature. It does not return a count. A key whose one-time keys are used up returns 0 and should be retired.
void	**wc_XmssKey_Free**(XmssKey * key)Releases resources held by an XmssKey. Safe to call with a NULL pointer. After this call the key is in state WC_XMSS_STATE_FREED and must be re-initialized before reuse.
int	**wc_XmssKey_ExportPub**(XmssKey * keyDst, const XmssKey * keySrc)Copies the public part of keySrc into keyDst. The destination key inherits the same parameter set and may be used for verification; it does not carry the private key state and cannot sign. Useful for handing a verifier the minimal data it needs.
int	**wc_XmssKey_ExportPub_ex**(XmssKey * keyDst, const XmssKey * keySrc, void * heap, int devId)Like wc_XmssKey_ExportPub() but the destination key is initialized fresh with the supplied heap and devId.
int	**wc_XmssKey_ExportPubRaw**(const XmssKey * key, byte * out, word32 * outLen)Exports the XMSS/XMSS^MT public key as a raw byte string. On entry *outLen is the size of out; on success it is updated to the bytes written.

	Name
int	**wc_XmssKey_ImportPubRaw**(XmssKey * key, const byte * in, word32 inLen)Imports a raw XMSS public key into key. The key must be in state WC_XMSS_STATE_INITED and the parameter set must already be selected (the raw encoding does NOT carry the parameter set, so the caller must apply it via wc_XmssKey_SetParamStr() first). On success the key transitions to state WC_XMSS_STATE_VERIFYONLY.
int	**wc_XmssKey_ImportPubRaw_ex**(XmssKey * key, const byte * in, word32 inLen, int is_xmssmt)Like wc_XmssKey_ImportPubRaw() but explicitly declares whether the encoded key is single_tree XMSS or multi_tree XMSS^MT (pass non_zero for XMSS^MT, zero for XMSS).
int	**wc_XmssKey_Verify**(XmssKey * key, const byte * sig, word32 sigSz, const byte * msg, int msgSz)Verifies an XMSS/XMSS^MT signature against msg using the public key held in key. The function returns 0 only when the signature is valid; any other value indicates the signature was rejected.

B.56.2 Detailed Description

XMSS (eXtended Merkle Signature Scheme) and its multi-tree variant XMSS^MT are stateful hash-based signature schemes specified in RFC 8391 and NIST SP 800-208. As with LMS, each signature consumes a one-time component of the private key, so the application MUST persist the private key state via the callbacks registered with wc_XmssKey_SetReadCb() and wc_XmssKey_SetWriteCb(). Reusing a one-time key destroys the security of the scheme.

The number of signatures available from a key is bounded by the parameter set; check whether any remain with wc_XmssKey_SigsLeft().

B.56.3 Functions Documentation

```
int wc_XmssKey_Init(
    XmssKey * key,
    void * heap,
    int devId
)
```

Initializes an XmssKey object. Must be called before any other XMSS/XMSS^MT operation. Use wc_XmssKey_Free() to release resources when done.

Parameters:

- **key** Pointer to the XmssKey to initialize.
- **heap** Heap hint for dynamic memory allocation. May be NULL.
- **devId** Device identifier for hardware crypto callbacks. Use INVALID_DEVID for software-only.

See:

- wc_XmssKey_Free
- wc_XmssKey_SetParamStr
- wc_XmssKey_MakeKey

Return:

- 0 on success.
- BAD_FUNC_ARG if key is NULL.

XMSS (eXtended Merkle Signature Scheme) and its multi-tree variant XMSS^MT (RFC 8391, NIST SP 800-208) are STATEFUL hash-based signature schemes: each call to wc_XmssKey_Sign() consumes a one_time component of the private key, and reusing a one_time key destroys the security of the scheme. Applications MUST persist the private key state between sign calls; see wc_XmssKey_SetWriteCb() and wc_XmssKey_SetReadCb().

After init the key is in state WC_XMSS_STATE_INITED. The parameter set must be selected by name with wc_XmssKey_SetParamStr() before generating or reloading a key.

Example

```
XmssKey key;
int ret;

ret = wc_XmssKey_Init(&key, NULL, INVALID_DEVID);
if (ret != 0) {
    // error initializing key
}
wc_XmssKey_SetParamStr(&key, "XMSS-SHA2_10_256");
// ... use key ...
wc_XmssKey_Free(&key);
```

```
int wc_XmssKey_InitId(
    XmssKey * key,
    const unsigned char * id,
    int len,
    void * heap,
    int devId
)
```

Initializes an XmssKey with a device-side key identifier. Equivalent to wc_XmssKey_Init() but also stashes a binary id that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.

Parameters:

- **key** Pointer to the XmssKey to initialize.
- **id** Pointer to the device-side key identifier bytes.
- **len** Number of bytes in id; must be in [0, XMSS_MAX_ID_LEN].
- **heap** Heap hint for dynamic memory allocation.

- **devId** Device identifier for the crypto callback.

See:

- wc_XmssKey_Init
- wc_XmssKey_InitLabel
- wc_XmssKey_Free

Return:

- 0 on success.
- BAD_FUNC_ARG if key is NULL, or id is NULL while len > 0.
- BUFFER_E if len is negative or greater than XMSS_MAX_ID_LEN.

The id is copied into the key object; the caller may free its buffer immediately after this call returns.

```
int wc_XmssKey_InitLabel(
    XmssKey * key,
    const char * label,
    void * heap,
    int devId
)
```

Initializes an XmssKey with a device-side key label. Equivalent to wc_XmssKey_Init() but also stashes a label string that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.

Parameters:

- **key** Pointer to the XmssKey to initialize.
- **label** NUL-terminated device-side key label.
- **heap** Heap hint for dynamic memory allocation.
- **devId** Device identifier for the crypto callback.

See:

- wc_XmssKey_Init
- wc_XmssKey_InitId

Return:

- 0 on success.
- BAD_FUNC_ARG if key or label is NULL.
- BUFFER_E if label is empty or longer than XMSS_MAX_LABEL_LEN.

```
int wc_XmssKey_SetParamStr(
    XmssKey * key,
    const char * str
)
```

Selects an XMSS or XMSS^MT parameter set by its RFC 8391 name. Accepted names take the form "XMSS-" (single-tree) or "XMSSMT-/" (multi-tree), for example "XMSS-SHA2_10_256" or "XMSSMT-SHA2_20/2_256". The set of names actually accepted depends on the hash families and tree heights enabled at build time.

Parameters:

- **key** Pointer to an XmssKey in state WC_XMSS_STATE_INITED.
- **str** Parameter set name (NUL-terminated).

See:

- wc_XmssKey_GetParamStr
- wc_XmssKey_MakeKey

Return:

- 0 on success.
- BAD_FUNC_ARG if key or str is NULL, or if the named parameter set is unknown or not compiled in.
- BAD_STATE_E if key is not in state WC_XMSS_STATE_INITED.

Example

```
XmssKey key;

wc_XmssKey_Init(&key, NULL, INVALID_DEVID);
wc_XmssKey_SetParamStr(&key, "XMSS-SHA2_10_256");
```

```
int wc_XmssKey_GetParamStr(
    const XmssKey * key,
    const char ** str
)
```

Retrieves the parameter set name currently selected on this key. The returned pointer is a static string and must not be freed by the caller.

Parameters:

- **key** Pointer to an XmssKey with a parameter set selected.
- **str** Receives a pointer to a static parameter name string.

See: wc_XmssKey_SetParamStr

Return:

- 0 on success.
- BAD_FUNC_ARG if key or str is NULL, or no parameter set has been selected.

```
int wc_XmssKey_SetWriteCb(
    XmssKey * key,
    wc_xmss_write_private_key_cb write_cb
)
```

Registers the callback that wolfSSL invokes to persist updated private key state. Because XMSS/XMSS^MT is stateful, the application MUST persist the private key after each successful sign before the signature is released; otherwise a crash or restart can lead to one-time key reuse and break the scheme.

Parameters:

- **key** Pointer to an XmssKey.
- **write_cb** Callback invoked to persist the private key.

See:

- wc_XmssKey_SetReadCb
- wc_XmssKey_SetContext
- wc_XmssKey_Sign

Return:

- 0 on success.
- BAD_FUNC_ARG if key or write_cb is NULL.

The callback returns one of the wc_XmssRc codes; WC_XMSS_RC_SAVED_TO_NV_MEMORY signals a durable write.

```
int wc_XmssKey_SetReadCb(
    XmssKey * key,
    wc_xmss_read_private_key_cb read_cb
)
```

Registers the callback that wolfSSL invokes to load persisted private key state. Used by wc_XmssKey_Reload() to bring a saved key back into memory for further signing.

Parameters:

- **key** Pointer to an XmssKey.
- **read_cb** Callback invoked to load the private key.

See:

- wc_XmssKey_SetWriteCb

- wc_XmssKey_Reload

Return:

- 0 on success.
- BAD_FUNC_ARG if key or read_cb is NULL.

```
int wc_XmssKey_SetContext(
    XmssKey * key,
    void * context
)
```

Sets the opaque context pointer passed to both the read and write private-key callbacks.

Parameters:

- **key** Pointer to an XmssKey.
- **context** Application-defined pointer; may be NULL.

See:

- wc_XmssKey_SetReadCb
- wc_XmssKey_SetWriteCb

Return:

- 0 on success.
- BAD_FUNC_ARG if key is NULL.

```
int wc_XmssKey_MakeKey(
    XmssKey * key,
    WC_RNG * rng
)
```

Generates a fresh XMSS/XMSS^MT key pair. The parameter set must already be selected via wc_XmssKey_SetParamStr() and the read/write callbacks must be registered. The newly generated private key is persisted via the write callback before the function returns; on success the key transitions to state WC_XMSS_STATE_OK.

Parameters:

- **key** Pointer to an XmssKey in state WC_XMSS_STATE_PARMSET with callbacks set.
- **rng** Pointer to an initialized WC_RNG.

See:

- wc_XmssKey_Sign

- wc_XmssKey_Reload

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- MEMORY_E on allocation failure.

Key generation can be slow for large tree heights; XMSS^MT variants amortize the cost over multiple smaller trees and generate noticeably faster than equivalent single-tree XMSS parameter sets.

Example

```
XmssKey key;
WC_RNG rng;

wc_XmssKey_Init(&key, NULL, INVALID_DEVID);
wc_XmssKey_SetParamStr(&key, "XMSS-SHA2_10_256");
wc_XmssKey_SetWriteCb(&key, my_write_cb);
wc_XmssKey_SetReadCb(&key, my_read_cb);
wc_XmssKey_SetContext(&key, &my_storage);
wc_InitRng(&rng);

if (wc_XmssKey_MakeKey(&key, &rng) != 0) {
    // error generating key
}
```

```
int wc_XmssKey_Reload(
    XmssKey * key
)
```

Reloads a previously generated XMSS/XMSS^MT private key from persistent storage using the registered read callback, restoring the key to a state where it can sign further messages. On success the key is in state WC_XMSS_STATE_OK.

Parameters:

- **key** Pointer to an XmssKey with parameters and read callback set.

See:

- wc_XmssKey_MakeKey
- wc_XmssKey_SetReadCb

Return:

- 0 on success, including the device-backed no-op.
- BAD_FUNC_ARG if any required pointer is NULL.
- WC_XMSS_RC_* mapped error if the read callback fails.
- IO_FAILED_E if the private key could not be read.

- NO_VALID_DEVID with WOLF_CRYPTO_CB_ONLY_XMSS when the key is not device-backed.

The same parameter set selected at key-generation time must be reapplied with wc_XmssKey_SetParamStr() before calling Reload.

With WOLF_CRYPTO_CB, a key on a devId with no read callback is taken to be device-backed and Reload is a no-op. With a read callback the software reload runs as usual, which needs the write callback set too.

WOLF_CRYPTO_CB_ONLY_XMSS removes the software reload, so the read callback is never called and only the device-backed no-op succeeds.

A reloaded key holds no public key: neither arm populates it. Call wc_XmssKey_ImportPubRaw() on a separate key to verify, or keep the public key exported at generation time.

```
int wc_XmssKey_GetPrivLen(
    const XmssKey * key,
    word32 * len
)
```

Returns the size in bytes of the encoded private key for the parameter set selected on this key.

Parameters:

- **key** Pointer to an XmssKey with parameters set.
- **len** Receives the private key size in bytes.

See:

- wc_XmssKey_GetPubLen
- wc_XmssKey_GetSigLen

Return:

- 0 on success.
- BAD_FUNC_ARG if key or len is NULL.

```
int wc_XmssKey_GetPubLen(
    const XmssKey * key,
    word32 * len
)
```

Returns the size in bytes of the XMSS/XMSS^MT public key for the parameter set selected on this key.

Parameters:

- **key** Pointer to an XmssKey with parameters set.
- **len** Receives the public key size in bytes.

See: wc_XmssKey_ExportPubRaw

Return:

- 0 on success.
- BAD_FUNC_ARG if key or len is NULL.

```
int wc_XmssKey_GetSigLen(
    const XmssKey * key,
    word32 * len
)
```

Returns the signature size in bytes for the parameter set selected on this key.

Parameters:

- **key** Pointer to an XmssKey with parameters set.
- **len** Receives the signature size in bytes.

See: wc_XmssKey_Sign

Return:

- 0 on success.
- BAD_FUNC_ARG if key or len is NULL.

```
int wc_XmssKey_Sign(
    XmssKey * key,
    byte * sig,
    word32 * sigSz,
    const byte * msg,
    int msgSz
)
```

Signs msg with the XMSS/XMSS^MT private key in key. On entry *sigSz is the size of the sig buffer; on success it is updated to the bytes written.

Parameters:

- **key** Pointer to an XmssKey in state WC_XMSS_STATE_OK.
- **sig** Buffer that receives the signature.
- **sigSz** In: size of sig. Out: bytes written.
- **msg** Message to sign.
- **msgSz** Length of msg in bytes.

See:

- wc_XmssKey_Verify

- wc_XmssKey_SigsLeft
- wc_XmssKey_SetWriteCb

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- BUFFER_E if *sigSz is smaller than the signature size.
- Negative error if all one-time keys have been used.

Each successful sign call consumes a one-time component of the private key. The updated key state is persisted via the registered write callback BEFORE the new signature is returned to the caller. If the write callback fails the sign call fails and the signature is not released. When the supply of one-time keys is exhausted the key transitions to state WC_XMSS_STATE_NOSIGS and further sign attempts fail – query wc_XmssKey_SigsLeft() to detect this condition in advance.

```
int wc_XmssKey_SigsLeft(
    XmssKey * key
)
```

Reports whether this key can create another signature. It does not return a count. A key whose one-time keys are used up returns 0 and should be retired.

Parameters:

- **key** Pointer to an XmssKey in state WC_XMSS_STATE_OK.

See: wc_XmssKey_Sign

Return:

- 1 if at least one signature remains.
- 0 if the one-time keys are exhausted, if key is NULL or not in state WC_XMSS_STATE_OK, if the read callback cannot load the private key, or if a crypto callback device returns an error. A key bound to a device that handles the request gets the device's answer.

```
void wc_XmssKey_Free(
    XmssKey * key
)
```

Releases resources held by an XmssKey. Safe to call with a NULL pointer. After this call the key is in state WC_XMSS_STATE_FREED and must be re-initialized before reuse.

Parameters:

- **key** Pointer to the XmssKey to free.

See: wc_XmssKey_Init

```
int wc_XmssKey_ExportPub(
    XmssKey * keyDst,
    const XmssKey * keySrc
)
```

Copies the public part of keySrc into keyDst. The destination key inherits the same parameter set and may be used for verification; it does not carry the private key state and cannot sign. Useful for handing a verifier the minimal data it needs.

Parameters:

- **keyDst** Pointer to an initialized destination XmssKey.
- **keySrc** Pointer to an XmssKey with the public key.

See:

- wc_XmssKey_ExportPub_ex
- wc_XmssKey_ExportPubRaw

Return:

- 0 on success.
- BAD_FUNC_ARG if keyDst or keySrc is NULL.
- BAD_STATE_E if keySrc holds no public key.

```
int wc_XmssKey_ExportPub_ex(
    XmssKey * keyDst,
    const XmssKey * keySrc,
    void * heap,
    int devId
)
```

Like wc_XmssKey_ExportPub() but the destination key is initialized fresh with the supplied heap and devId.

Parameters:

- **keyDst** Pointer to an XmssKey to populate.
- **keySrc** Pointer to an XmssKey with the public key.
- **heap** Heap hint for keyDst.
- **devId** Device identifier for keyDst.

See: wc_XmssKey_ExportPub

Return:

- 0 on success.
- BAD_FUNC_ARG if keyDst or keySrc is NULL.
- BAD_STATE_E if keySrc holds no public key.

```
int wc_XmssKey_ExportPubRaw(
    const XmssKey * key,
    byte * out,
    word32 * outLen
)
```

Exports the XMSS/XMSS^MT public key as a raw byte string. On entry *outLen is the size of out; on success it is updated to the bytes written.

Parameters:

- **key** Pointer to an XmssKey.
- **out** Buffer that receives the public key.
- **outLen** In: size of out. Out: bytes written.

See:

- wc_XmssKey_ImportPubRaw
- wc_XmssKey_GetPubLen

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- BAD_STATE_E if key holds no public key.
- BUFFER_E if *outLen is smaller than the public key size.

```
int wc_XmssKey_ImportPubRaw(
    XmssKey * key,
    const byte * in,
    word32 inLen
)
```

Imports a raw XMSS public key into key. The key must be in state WC_XMSS_STATE_INITED and the parameter set must already be selected (the raw encoding does NOT carry the parameter set, so the caller must apply it via wc_XmssKey_SetParamStr() first). On success the key transitions to state WC_XMSS_STATE_VERIFYONLY.

Parameters:

- **key** Pointer to an XmssKey with a parameter set.
- **in** Raw public key bytes.
- **inLen** Length of in in bytes.

See:

- wc_XmssKey_ImportPubRaw_ex
- wc_XmssKey_ExportPubRaw
- wc_XmssKey_Verify

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- BUFFER_E if inLen does not match the expected public key size.

```
int wc_XmssKey_ImportPubRaw_ex(
    XmssKey * key,
    const byte * in,
    word32 inLen,
    int is_xmssmt
)
```

Like wc_XmssKey_ImportPubRaw() but explicitly declares whether the encoded key is single_tree XMSS or multi_tree XMSS^MT (pass non_zero for XMSS^MT, zero for XMSS).

Parameters:

- **key** Pointer to an XmssKey with a parameter set.
- **in** Raw public key bytes.
- **inLen** Length of in in bytes.
- **is_xmssmt** Non-zero if the key is XMSS^MT, zero if plain XMSS.

See: wc_XmssKey_ImportPubRaw

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- BUFFER_E if inLen does not match the expected public key size.

```
int wc_XmssKey_Verify(
    XmssKey * key,
    const byte * sig,
    word32 sigSz,
    const byte * msg,
    int msgSz
)
```

Verifies an XMSS/XMSS^MT signature against msg using the public key held in key. The function returns 0 only when the signature is valid; any other value indicates the signature was rejected.

Parameters:

- **key** Pointer to an XmssKey with the public key.
- **sig** Signature bytes to verify.
- **sigSz** Length of sig in bytes.
- **msg** Message that was signed.

- **msgSz** Length of msg in bytes.

See:

- wc_XmssKey_Sign
- wc_XmssKey_ImportPubRaw

Return:

- 0 on a valid signature.
- BAD_FUNC_ARG if any required pointer is NULL.
- BAD_STATE_E if key holds no public key.
- SIG_VERIFY_E (or similar) if the signature is invalid or malformed.

C API Header Files

C.1 aes.h

C.1.1 Functions

	Name
int	**wc_AesSetKey**(Aes * aes, const byte * key, word32 len, const byte * iv, int dir)This function initializes an AES structure by setting the key and then setting the initialization vector.
int	**wc_AesSetIV**(Aes * aes, const byte * iv)This function sets the initialization vector for a particular AES object. The AES object should be initialized before calling this function.
int	**wc_AesCbcEncrypt**(Aes * aes, byte * out, const byte * in, word32 sz)Encrypts a plaintext message from the input buffer in, and places the resulting cipher text in the output buffer out using cipher block chaining with AES. This function requires that the AES object has been initialized by calling AesSetKey before a message is able to be encrypted. This function assumes that the input message is AES block length aligned, and expects the input length to be a multiple of the block length, which will optionally be checked and enforced if WOLFSSL_AES_CBC_LENGTH_CHECKS is defined in the build configuration. In order to assure block-multiple input, PKCS#7 style padding should be added beforehand. This differs from the OpenSSL AES-CBC methods which add the padding for you. To make the wolfSSL and corresponding OpenSSL functions interoperate, one should specify the -nopad option in the OpenSSL command line function so that it behaves like the wolfSSL AesCbcEncrypt method and does not add extra padding during encryption.

	Name
int	**wc_AesCbcDecrypt**(Aes * aes, byte * out, const byte * in, word32 sz)Decrypts a cipher from the input buffer in, and places the resulting plain text in the output buffer out using cipher block chaining with AES. This function requires that the AES structure has been initialized by calling AesSetKey before a message is able to be decrypted. This function assumes that the original message was AES block length aligned, and expects the input length to be a multiple of the block length, which will optionally be checked and enforced if WOLFSSL_AES_CBC_LENGTH_CHECKS is defined in the build configuration. This differs from the OpenSSL AES-CBC methods, which add PKCS#7 padding automatically, and so do not require block-multiple input. To make the wolfSSL function and equivalent OpenSSL functions interoperate, one should specify the -nopad option in the OpenSSL command line function so that it behaves like the wolfSSL AesCbcEncrypt method and does not create errors during decryption.
int	**wc_AesCtrEncrypt**(Aes * aes, byte * out, const byte * in, word32 sz)Encrypts/Decrypts a message from the input buffer in, and places the resulting cipher text in the output buffer out using CTR mode with AES. This function is only enabled if WOLFSSL_AES_COUNTER is enabled at compile time. The AES structure should be initialized through AesSetKey before calling this function. Note that this function is used for both decryption and encryption. *NOTE:* Regarding using same API for encryption and decryption. User should differentiate between Aes structures for encrypt/decrypt.

	Name
int	**wc_AesEncryptDirect**(Aes * aes, byte * out, const byte * in)This function is a one_block encrypt of the input block, in, into the output block, out. It uses the key of the provided AES structure, which should be initialized with wc_AesSetKey before calling this function. wc_AesSetKey should have been called with the iv set to NULL. This is only enabled if the configure option WOLFSSL_AES_DIRECT is enabled. **Warning:** In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB API's directly whenever possible. When WC_AES_DIRECT_CRYPTO_CB is defined (set automatically for WOLF_CRYPTO_CB builds that use the generic AES Direct code, not a port's own version), if the AES object has a crypto callback device (devId), that device is asked first to encrypt the block as a one-block ECB request. The software key is used only if the device declines, and a device error is returned as-is. A device that owns the key must handle the request or return an error; it must not return CRYPTOCB_UNAVAILABLE. With WOLF_CRYPTO_CB_FIND, the device find callback picks the device, so a device can be asked even when the AES object has no devId.
int	**wc_AesDecryptDirect**(Aes * aes, byte * out, const byte * in)This function is a one_block decrypt of the input block, in, into the output block, out. It uses the key of the provided AES structure, which should be initialized with wc_AesSetKey before calling this function. wc_AesSetKey should have been called with the iv set to NULL. This is only enabled if the configure option WOLFSSL_AES_DIRECT is enabled. **Warning:** In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB API's directly whenever possible. When WC_AES_DIRECT_CRYPTO_CB is defined (set automatically for WOLF_CRYPTO_CB builds that use the generic AES Direct code, not a port's own version), if the AES object has a crypto callback device (devId), that device is asked first to decrypt the block as a one-block ECB request. The software key is used only if the device declines, and a device error is returned as-is. A device that owns the key must handle the request or return an error; it must not return CRYPTOCB_UNAVAILABLE. With WOLF_CRYPTO_CB_FIND, the device find callback picks the device, so a device can be asked even when the AES object has no devId.

	Name
int	**wc_AesSetKeyDirect**(Aes * aes, const byte * key, word32 len, const byte * iv, int dir)This function is used to set the AES keys for CTR mode with AES. It initializes an AES object with the given key, iv (initialization vector), and encryption dir (direction). It is only enabled if the configure option WOLFSSL_AES_DIRECT is enabled. Currently wc_AesSetKeyDirect uses wc_AesSetKey internally. **Warning:** In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB API's directly whenever possible.
int	**wc_AesGcmSetKey**(Aes * aes, const byte * key, word32 len)This function is used to set the key for AES GCM (Galois/Counter Mode). It initializes an AES object with the given key. It is only enabled if the configure option HAVE_AESGCM is enabled at compile time.
int	**wc_AesSetTagLen**(Aes * aes, word32 tagLen)This function associates one tag length with the key held by an AES object, as SP 800_38D section 5.2.1.2 and SP 800_38C section 5.3 require. Once set, AES_GCM and AES_CCM calls with that key accept only that tag length. wc_AesGcmSetKey() and wc_AesCcmSetKey() clear the association. The mode still applies its own list of allowed tag lengths on top of this.
int	**wc_AesGcmEncrypt**(Aes * aes, byte * out, const byte * in, word32 sz, const byte * iv, word32 ivSz, byte * authTag, word32 authTagSz, const byte * authIn, word32 authInSz)This function encrypts the input message, held in the buffer in, and stores the resulting cipher text in the output buffer out. It requires a new iv (initialization vector) for each call to encrypt. It also encodes the input authentication vector, authIn, into the authentication tag, authTag.
int	**wc_AesGcmDecrypt**(Aes * aes, byte * out, const byte * in, word32 sz, const byte * iv, word32 ivSz, const byte * authTag, word32 authTagSz, const byte * authIn, word32 authInSz)This function decrypts the input cipher text, held in the buffer in, and stores the resulting message text in the output buffer out. It also checks the input authentication vector, authIn, against the supplied authentication tag, authTag. If a nonzero error code is returned, the output data is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.

	Name
int	**wc_GmacSetKey**(Gmac * gmac, const byte * key, word32 len)This function initializes and sets the key for a GMAC object to be used for Galois Message Authentication.
int	**wc_GmacUpdate**(Gmac * gmac, const byte * iv, word32 ivSz, const byte * authIn, word32 authInSz, byte * authTag, word32 authTagSz)This function generates the Gmac hash of the authIn input and stores the result in the authTag buffer. After running wc_GmacUpdate, one should compare the generated authTag to a known authentication tag to verify the authenticity of a message.
int	**wc_AesCcmSetKey**(Aes * aes, const byte * key, word32 keySz)This function sets the key for an AES object using CCM (Counter with CBC_MAC). It takes a pointer to an AES structure and initializes it with supplied key.
int	**wc_AesCcmEncrypt**(Aes * aes, byte * out, const byte * in, word32 inSz, const byte * nonce, word32 nonceSz, byte * authTag, word32 authTagSz, const byte * authIn, word32 authInSz)This function encrypts the input message, in, into the output buffer, out, using CCM (Counter with CBC_MAC). It subsequently calculates and stores the authorization tag, authTag, from the authIn input.
int	**wc_AesCcmDecrypt**(Aes * aes, byte * out, const byte * in, word32 inSz, const byte * nonce, word32 nonceSz, const byte * authTag, word32 authTagSz, const byte * authIn, word32 authInSz)This function decrypts the input cipher text, in, into the output buffer, out, using CCM (Counter with CBC_MAC). It subsequently calculates the authorization tag, authTag, from the authIn input. If a nonzero error code is returned, the output data is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.
int	**wc_AesXtsInit**(XtsAes * aes, void * heap, int devId)This is to initialize an AES-XTS context. It is up to user to call wc_AesXtsFree on aes key when done.
int	**wc_AesXtsSetKeyNoInit**(XtsAes * aes, const byte * key, word32 len, int dir)This is to help with setting keys to correct encrypt or decrypt type, after first calling wc_AesXtsInit(). It is up to user to call wc_AesXtsFree on aes key when done.

	Name
int	**wc_AesXtsSetKey**(XtsAes * aes, const byte * key, word32 len, int dir, void * heap, int devId)This is to help with setting keys to correct encrypt or decrypt type. It is up to user to call wc_AesXtsFree on aes key when done.
int	**wc_AesXtsEncryptSector**(XtsAes * aes, byte * out, const byte * in, word32 sz, word64 sector)Same process as wc_AesXtsEncrypt but uses a word64 type as the tweak value instead of a byte array. This just converts the word64 to a byte array and calls wc_AesXtsEncrypt.
int	**wc_AesXtsDecryptSector**(XtsAes * aes, byte * out, const byte * in, word32 sz, word64 sector)Same process as wc_AesXtsDecrypt but uses a word64 type as the tweak value instead of a byte array. This just converts the word64 to a byte array.
int	**wc_AesXtsEncrypt**(XtsAes * aes, byte * out, const byte * in, word32 sz, const byte * i, word32 iSz)AES with XTS mode. (XTS) XEX encryption with Tweak and cipher text Stealing.
int	**wc_AesXtsDecrypt**(XtsAes * aes, byte * out, const byte * in, word32 sz, const byte * i, word32 iSz)Same process as encryption but Aes key is AES_DECRYPTION type.
int	**wc_AesXtsFree**(XtsAes * aes)This is to free up any resources used by the XtsAes structure.
int	**wc_AesInit**(Aes * aes, void * heap, int devId)Initialize Aes structure. Sets heap hint to be used and ID for use with async hardware. It is up to the user to call wc_AesFree on the Aes structure when done.
void	**wc_AesFree**(Aes * aes)free resources associated with the Aes structure when applicable. Internally may sometimes be a no_op but still recommended to call in all cases as a general best_practice (IE if application code is ported for use on new environments where the call is applicable).
int	**wc_AesCfbEncrypt**(Aes * aes, byte * out, const byte * in, word32 sz)AES with CFB mode.
int	**wc_AesCfbDecrypt**(Aes * aes, byte * out, const byte * in, word32 sz)AES with CFB mode.
int	**wc_AesSivEncrypt**(const byte * key, word32 keySz, const byte * assoc, word32 assocSz, const byte * nonce, word32 nonceSz, const byte * in, word32 inSz, byte * siv, byte * out)This function performs SIV (synthetic initialization vector) encryption as described in RFC 5297.

	Name
int	**wc_AesSivDecrypt**(const byte * key, word32 keySz, const byte * assoc, word32 assocSz, const byte * nonce, word32 nonceSz, const byte * in, word32 inSz, byte * siv, byte * out)This function performs SIV (synthetic initialization vector) decryption as described in RFC 5297. If a nonzero error code is returned, the output data is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.
WOLFSSL_API int	**wc_AesEaxEncryptAuth**(const byte * key, word32 keySz, byte * out, const byte * in, word32 inSz, const byte * nonce, word32 nonceSz, byte * authTag, word32 authTagSz, const byte * authIn, word32 authInSz)This function performs AES EAX encryption and authentication as described in “EAX: A Conventional Authenticated_Encryption Mode” (https://eprint.iacr.org/2003/069). It is a “one-shot” API that performs all encryption and authentication operations in one function call.
WOLFSSL_API int	**wc_AesEaxDecryptAuth**(const byte * key, word32 keySz, byte * out, const byte * in, word32 inSz, const byte * nonce, word32 nonceSz, const byte * authTag, word32 authTagSz, const byte * authIn, word32 authInSz)This function performs AES EAX decryption and authentication as described in “EAX: A Conventional Authenticated_Encryption Mode” (https://eprint.iacr.org/2003/069). It is a “one-shot” API that performs all decryption and authentication operations in one function call. If a nonzero error code is returned, the output data is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.
WOLFSSL_API int	**wc_AesEaxInit**(AesEax * eax, const byte * key, word32 keySz, const byte * nonce, word32 nonceSz, const byte * authIn, word32 authInSz)This function initializes an AesEax object for use in authenticated encryption or decryption. This function must be called on an AesEax object before using it with any of the AES EAX incremental API functions. It does not need to be called if using the one_shot EAX API functions. All AesEax instances initialized with this function need to be freed with a call to wc_AesEaxFree() when done using the instance.

	Name
WOLFSSL_API int	**wc_AesEaxEncryptUpdate**(AesEax * eax, byte * out, const byte * in, word32 inSz, const byte * authIn, word32 authInSz)This function uses AES EAX to encrypt input data, and optionally, add more input data to the authentication stream. eax must have been previously initialized with a call to wc_AesEaxInit.
WOLFSSL_API int	**wc_AesEaxDecryptUpdate**(AesEax * eax, byte * out, const byte * in, word32 inSz, const byte * authIn, word32 authInSz)This function uses AES EAX to decrypt input data, and optionally, add more input data to the authentication stream. eax must have been previously initialized with a call to wc_AesEaxInit.
WOLFSSL_API int	**wc_AesEaxAuthDataUpdate**(AesEax * eax, const byte * authIn, word32 authInSz)This function adds input data to the authentication stream. eax must have been previously initialized with a call to wc_AesEaxInit.
WOLFSSL_API int	**wc_AesEaxEncryptFinal**(AesEax * eax, byte * authTag, word32 authTagSz)This function finalizes the encrypt AEAD operation, producing an auth tag over the current authentication stream. eax must have been previously initialized with a call to wc_AesEaxInit. When done using the AesEax context structure, make sure to free it using wc_AesEaxFree.
WOLFSSL_API int	**wc_AesEaxDecryptFinal**(AesEax * eax, const byte * authIn, word32 authInSz)This function finalizes the decrypt AEAD operation, finalizing the auth tag computation and checking it for validity against the user supplied tag. eax must have been previously initialized with a call to wc_AesEaxInit. When done using the AesEax context structure, make sure to free it using wc_AesEaxFree.
WOLFSSL_API int	**wc_AesEaxFree**(AesEax * eax)This frees up any resources, specifically keys, used by the Aes instance inside the AesEax wrapper struct. It should be called on the AesEax struct after it has been initialized with wc_AesEaxInit, and all desired EAX operations are complete.
int	**wc_AesCtsEncrypt**(const byte * key, word32 keySz, byte * out, const byte * in, word32 inSz, const byte * iv)This function performs AES encryption using Ciphertext Stealing (CTS) mode. It is a one-shot API that handles all operations in a single call.

	Name
int	**wc_AesCtsDecrypt**(const byte * key, word32 keySz, byte * out, const byte * in, word32 inSz, const byte * iv)This function performs AES decryption using Ciphertext Stealing (CTS) mode. It is a one-shot API that handles all operations in a single call.
int	**wc_AesCtsEncryptUpdate**(Aes * aes, byte * out, word32 * outSz, const byte * in, word32 inSz)This function performs an update step of the AES CTS encryption. It processes a chunk of plaintext and stores intermediate data.
int	**wc_AesCtsEncryptFinal**(Aes * aes, byte * out, word32 * outSz)This function finalizes the AES CTS encryption operation. It processes any remaining plaintext and completes the encryption.
int	**wc_AesCtsDecryptUpdate**(Aes * aes, byte * out, word32 * outSz, const byte * in, word32 inSz)This function performs an update step of the AES CTS decryption. It processes a chunk of ciphertext and stores intermediate data.
int	**wc_AesCtsDecryptFinal**(Aes * aes, byte * out, word32 * outSz)This function finalizes the AES CTS decryption operation. It processes any remaining ciphertext and completes the decryption.
int	**wc_AesCfb1Encrypt**(Aes * aes, byte * out, const byte * in, word32 sz)This function encrypts data using AES CFB_1 mode (1_bit feedback). It processes data one bit at a time, making it suitable for bit-oriented applications.
int	**wc_AesCfb8Encrypt**(Aes * aes, byte * out, const byte * in, word32 sz)This function encrypts data using AES CFB_8 mode (8_bit feedback). It processes data one byte at a time, making it suitable for byte-oriented stream encryption.
int	**wc_AesCfb1Decrypt**(Aes * aes, byte * out, const byte * in, word32 sz)This function decrypts data using AES CFB_1 mode (1_bit feedback). It processes data one bit at a time, making it suitable for bit-oriented applications.
int	**wc_AesCfb8Decrypt**(Aes * aes, byte * out, const byte * in, word32 sz)This function decrypts data using AES CFB_8 mode (8_bit feedback). It processes data one byte at a time, making it suitable for byte-oriented stream decryption.

	Name
int	**wc_AesOfbEncrypt**(Aes * aes, byte * out, const byte * in, word32 sz)This function encrypts data using AES OFB mode (Output Feedback). OFB mode turns a block cipher into a stream cipher by encrypting the IV and XORing with plaintext.
int	**wc_AesOfbDecrypt**(Aes * aes, byte * out, const byte * in, word32 sz)This function decrypts data using AES OFB mode (Output Feedback). In OFB mode, encryption and decryption are the same operation.
int	**wc_AesEcbEncrypt**(Aes * aes, byte * out, const byte * in, word32 sz)This function encrypts data using AES ECB mode (Electronic Codebook). Warning: ECB mode is not recommended for most use cases as it does not provide semantic security. Each block is encrypted independently.
int	**wc_AesEcbDecrypt**(Aes * aes, byte * out, const byte * in, word32 sz)This function decrypts data using AES ECB mode (Electronic Codebook). Warning: ECB mode is not recommended for most use cases as it does not provide semantic security. Each block is decrypted independently.
int	**wc_AesCtrSetKey**(Aes * aes, const byte * key, word32 len, const byte * iv, int dir)This function sets the key and IV for AES CTR mode. It initializes the AES structure for counter mode encryption or decryption.
int	**wc_AesGcmSetKey_ex**(Aes * aes, const byte * key, word32 len, word32 kup)This function sets the key for AES GCM with an extended key update parameter. It allows for key updates in certain hardware implementations.
int	**wc_AesGcmInit**(Aes * aes, const byte * key, word32 len, const byte * iv, word32 ivSz)This function initializes an AES GCM cipher with key and IV. It can be called with NULL key to only set the IV, or with NULL IV to only set the key.
int	**wc_AesGcmEncryptInit**(Aes * aes, const byte * key, word32 len, const byte * iv, word32 ivSz)This function initializes an AES GCM cipher for encryption. It is a convenience wrapper around wc_AesGcmInit for encryption operations.
int	**wc_AesGcmEncryptInit_ex**(Aes * aes, const byte * key, word32 len, byte * ivOut, word32 ivOutSz)This function initializes an AES GCM cipher for encryption and outputs the IV. This is useful when part of the IV is generated internally. Must call wc_AesGcmSetIV() before this function to set the fixed part of the IV.

	Name
int	**wc_AesGcmEncryptUpdate**(Aes * aes, byte * out, const byte * in, word32 sz, const byte * authIn, word32 authInSz)This function performs an update step of AES GCM encryption. It processes plaintext and/or additional authentication data (AAD) in a streaming fashion.
int	**wc_AesGcmEncryptFinal**(Aes * aes, byte * authTag, word32 authTagSz)This function finalizes AES GCM encryption and generates the authentication tag. This must be called after all data has been processed with wc_AesGcmEncryptUpdate.
int	**wc_AesGcmDecryptInit**(Aes * aes, const byte * key, word32 len, const byte * iv, word32 ivSz)This function initializes an AES GCM cipher for decryption. It is a convenience wrapper around wc_AesGcmInit for decryption operations.
int	**wc_AesGcmDecryptUpdate**(Aes * aes, byte * out, const byte * in, word32 sz, const byte * authIn, word32 authInSz)This function performs an update step of AES GCM decryption. It processes ciphertext and/or additional authentication data (AAD) in a streaming fashion.
int	**wc_AesGcmDecryptFinal**(Aes * aes, const byte * authTag, word32 authTagSz)This function finalizes AES GCM decryption and verifies the authentication tag. This must be called after all data has been processed with wc_AesGcmDecryptUpdate.
int	**wc_AesGcmSetExtIV**(Aes * aes, const byte * iv, word32 ivSz)This function sets an external IV for AES GCM. This allows using an IV that was generated externally or received from another source.
int	**wc_AesGcmSetIV**(Aes * aes, word32 ivSz, const byte * ivFixed, word32 ivFixedSz, WC_RNG * rng)This function sets the IV for AES GCM with optional random generation. It can generate part of the IV using an RNG, which is useful for ensuring IV uniqueness.
int	**wc_AesGcmEncrypt_ex**(Aes * aes, byte * out, const byte * in, word32 sz, byte * ivOut, word32 ivOutSz, byte * authTag, word32 authTagSz, const byte * authIn, word32 authInSz)This function performs AES GCM encryption with extended parameters, including IV output. This is a one-shot encryption function that outputs the generated IV.

	Name
int	**wc_Gmac**(const byte * key, word32 keySz, byte * iv, word32 ivSz, const byte * authIn, word32 authInSz, byte * authTag, word32 authTagSz, WC_RNG * rng)This function performs GMAC (Galois Message Authentication Code) generation. GMAC is essentially AES-GCM with no plaintext, used for authentication only.
int	**wc_GmacVerify**(const byte * key, word32 keySz, const byte * iv, word32 ivSz, const byte * authIn, word32 authInSz, const byte * authTag, word32 authTagSz)This function verifies a GMAC (Galois Message Authentication Code). It computes the GMAC and compares it with the provided tag.
int	**wc_AesCcmSetNonce**(Aes * aes, const byte * nonce, word32 nonceSz)This function sets the nonce for AES CCM mode. The nonce must be set before encryption or decryption operations.
int	**wc_AesCcmEncrypt_ex**(Aes * aes, byte * out, const byte * in, word32 sz, byte * ivOut, word32 ivOutSz, byte * authTag, word32 authTagSz, const byte * authIn, word32 authInSz)This function performs AES CCM encryption with extended parameters, including nonce output. This is useful when part of the nonce is generated internally.
int	**wc_AesKeyWrap**(const byte * key, word32 keySz, const byte * in, word32 inSz, byte * out, word32 outSz, const byte * iv)This function wraps a key using AES Key Wrap algorithm (RFC 3394). This is commonly used to securely transport cryptographic keys.
int	**wc_AesKeyWrap_ex**(Aes * aes, const byte * in, word32 inSz, byte * out, word32 outSz, const byte * iv)This function wraps a key using AES Key Wrap algorithm with an initialized AES structure. This allows reusing the same AES structure for multiple wrap operations.
int	**wc_AesKeyUnWrap**(const byte * key, word32 keySz, const byte * in, word32 inSz, byte * out, word32 outSz, const byte * iv)This function unwraps a key using AES Key Unwrap algorithm (RFC 3394). This is used to securely receive cryptographic keys that were wrapped.
int	**wc_AesKeyUnWrap_ex**(Aes * aes, const byte * in, word32 inSz, byte * out, word32 outSz, const byte * iv)This function unwraps a key using AES Key Unwrap algorithm with an initialized AES structure. This allows reusing the same AES structure for multiple unwrap operations.

	Name
int	**wc_AesXtsEncryptConsecutiveSectors**(XtsAes * aes, byte * out, const byte * in, word32 sz, word64 sector, word32 sectorSz)This function encrypts multiple consecutive sectors using AES XTS mode. It processes multiple sectors in sequence, automatically incrementing the sector number for each sector.
int	**wc_AesXtsDecryptConsecutiveSectors**(XtsAes * aes, byte * out, const byte * in, word32 sz, word64 sector, word32 sectorSz)This function decrypts multiple consecutive sectors using AES XTS mode. It processes multiple sectors in sequence, automatically incrementing the sector number for each sector.
int	**wc_AesXtsEncryptInit**(XtsAes * aes, const byte * i, word32 iSz, struct XtsAesStreamData * stream)This function initializes streaming AES XTS encryption. It sets up the context for processing data in multiple update calls.
int	**wc_AesXtsDecryptInit**(XtsAes * aes, const byte * i, word32 iSz, struct XtsAesStreamData * stream)This function initializes streaming AES XTS decryption. It sets up the context for processing data in multiple update calls.
int	**wc_AesXtsEncryptUpdate**(XtsAes * aes, byte * out, const byte * in, word32 sz, struct XtsAesStreamData * stream)This function performs an update step of streaming AES XTS encryption. It processes a chunk of data and can be called multiple times.
int	**wc_AesXtsDecryptUpdate**(XtsAes * aes, byte * out, const byte * in, word32 sz, struct XtsAesStreamData * stream)This function performs an update step of streaming AES XTS decryption. It processes a chunk of data and can be called multiple times.
int	**wc_AesXtsEncryptFinal**(XtsAes * aes, byte * out, const byte * in, word32 sz, struct XtsAesStreamData * stream)This function finalizes streaming AES XTS encryption. It processes any remaining data and completes the encryption operation.
int	**wc_AesXtsDecryptFinal**(XtsAes * aes, byte * out, const byte * in, word32 sz, struct XtsAesStreamData * stream)This function finalizes streaming AES XTS decryption. It processes any remaining data and completes the decryption operation.
int	**wc_AesGetKeySize**(Aes * aes, word32 * keySize)This function retrieves the key size from an initialized AES structure. It returns the size of the key currently set in the AES object.

	Name
int	**wc_AesInit_Id**(Aes * aes, unsigned char * id, int len, void * heap, int devId)This function initializes an AES structure with an ID. This is useful for tracking or identifying specific AES instances in applications that manage multiple AES contexts.
int	**wc_AesInit_Label**(Aes * aes, const char * label, void * heap, int devId)This function initializes an AES structure with a label string. This is useful for tracking or identifying specific AES instances with human-readable names.
Aes *	**wc_AesNew**(void * heap, int devId, int * result_code)This function allocates and initializes a new AES structure. It returns a pointer to the allocated structure, which must be freed with wc_AesDelete when no longer needed. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.
int	**wc_AesDelete**(Aes * aes, Aes ** aes_p)This function frees an AES structure that was allocated with wc_AesNew. It also sets the pointer to NULL to prevent use-after-free. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.
int	**wc_AesSivEncrypt_ex**(const byte * key, word32 keySz, const AesSivAssoc * assoc, word32 numAssoc, const byte * nonce, word32 nonceSz, const byte * in, word32 inSz, byte * siv, byte * out)This function performs AES_SIV (Synthetic IV) encryption with extended parameters. AES-SIV provides nonce-misuse resistance and deterministic authenticated encryption.
int	**wc_AesSivDecrypt_ex**(const byte * key, word32 keySz, const AesSivAssoc * assoc, word32 numAssoc, const byte * nonce, word32 nonceSz, const byte * in, word32 inSz, byte * siv, byte * out)This function performs AES_SIV (Synthetic IV) decryption with extended parameters. It verifies the SIV and decrypts the ciphertext.

C.1.2 Functions Documentation

C.1.2.1 function wc_AesSetKey

```
int wc_AesSetKey(
    Aes * aes,
```

```
    const byte * key,
    word32 len,
    const byte * iv,
    int dir
)
```

This function initializes an AES structure by setting the key and then setting the initialization vector.

Parameters:

- **aes** pointer to the AES structure to modify
- **key** 16, 24, or 32 byte secret key for encryption and decryption
- **len** length of the key passed in
- **iv** pointer to the initialization vector used to initialize the key
- **dir** Cipher direction. Set AES_ENCRYPTION to encrypt, or AES_DECRYPTION to decrypt. Direction for some modes (CFB and CTR) is always AES_ENCRYPTION.

See:

- wc_AesSetKeyDirect
- wc_AesSetIV

Return:

- 0 On successfully setting key and initialization vector.
- BAD_FUNC_ARG Returned if key length is invalid.

Example

```
Aes enc;
int ret = 0;
byte key[] = { some 16, 24 or 32 byte key };
byte iv[]  = { some 16 byte iv };
if (ret = wc_AesInit(&enc, HEAP_HINT, INVALID_DEVID) != 0) {
    // failed to initialize aes key
}
if (ret = wc_AesSetKey(&enc, key, AES_BLOCK_SIZE, iv,
AES_ENCRYPTION) != 0) {
// failed to set aes key
}
```

C.1.2.2 function wc_AesSetIV

```
int wc_AesSetIV(
    Aes * aes,
    const byte * iv
)
```

This function sets the initialization vector for a particular AES object. The AES object should be initialized before calling this function.

Parameters:

- **aes** pointer to the AES structure on which to set the initialization vector
- **iv** initialization vector used to initialize the AES structure. If the value is NULL, the default action initializes the iv to 0.

See:

- wc_AesSetKeyDirect
- wc_AesSetKey

Return:

- 0 On successfully setting initialization vector.
- BAD_FUNC_ARG Returned if AES pointer is NULL.

Example

```
Aes enc;
// set enc key
byte iv[]  = { some 16 byte iv };
if (ret = wc_AesSetIV(&enc, iv) != 0) {
// failed to set aes iv
}
```

C.1.2.3 function wc_AesCbcEncrypt

```
int wc_AesCbcEncrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

Encrypts a plaintext message from the input buffer in, and places the resulting cipher text in the output buffer out using cipher block chaining with AES. This function requires that the AES object has been initialized by calling AesSetKey before a message is able to be encrypted. This function assumes that the input message is AES block length aligned, and expects the input length to be a multiple of the block length, which will optionally be checked and enforced if WOLFSSL_AES_CBC_LENGTH_CHECKS is defined in the build configuration. In order to assure block-multiple input, PKCS#7 style padding should be added beforehand. This differs from the OpenSSL AES-CBC methods which add the padding for you. To make the wolfSSL and corresponding OpenSSL functions interoperate, one should specify the -nopad option in the OpenSSL command line function so that it behaves like the wolfSSL AesCbcEncrypt method and does not add extra padding during encryption.

Parameters:

- **aes** pointer to the AES object used to encrypt data
- **out** pointer to the output buffer in which to store the ciphertext of the encrypted message
- **in** pointer to the input buffer containing message to be encrypted
- **sz** size of input message

See:

- wc_AesInit
- wc_AesSetKey
- wc_AesSetIV
- wc_AesCbcDecrypt

Return:

- 0 On successfully encrypting message.
- BAD_ALIGN_E: may be returned on block align error
- BAD_LENGTH_E will be returned if the input length isn't a multiple of the AES block length, when the library is built with WOLFSSL_AES_CBC_LENGTH_CHECKS.

Example

```
Aes enc;
int ret = 0;
// initialize enc with wc_AesInit and wc_AesSetKey, using direction
```

```
// AES_ENCRYPTION
byte msg[AES_BLOCK_SIZE * n]; // multiple of 16 bytes
// fill msg with data
byte cipher[AES_BLOCK_SIZE * n]; // Some multiple of 16 bytes
if ((ret = wc_AesCbcEncrypt(&enc, cipher, message, sizeof(msg))) != 0 ) {
// block align error
}
```

C.1.2.4 function wc_AesCbcDecrypt

```
int wc_AesCbcDecrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

Decrypts a cipher from the input buffer in, and places the resulting plain text in the output buffer out using cipher block chaining with AES. This function requires that the AES structure has been initialized by calling AesSetKey before a message is able to be decrypted. This function assumes that the original message was AES block length aligned, and expects the input length to be a multiple of the block length, which will optionally be checked and enforced if WOLFSSL_AES_CBC_LENGTH_CHECKS is defined in the build configuration. This differs from the OpenSSL AES-CBC methods, which add PKCS#7 padding automatically, and so do not require block-multiple input. To make the wolfSSL function and equivalent OpenSSL functions interoperate, one should specify the -nopad option in the OpenSSL command line function so that it behaves like the wolfSSL AesCbcEncrypt method and does not create errors during decryption.

Parameters:

- **aes** pointer to the AES object used to decrypt data.
- **out** pointer to the output buffer in which to store the plain text of the decrypted message. size must be a multiple of AES_BLOCK_LENGTH, padded if necessary
- **in** pointer to the input buffer containing cipher text to be decrypted. size must be a multiple of AES_BLOCK_LENGTH, padded if necessary
- **sz** size of input message.

See:

- wc_AesInit
- wc_AesSetKey
- wc_AesCbcEncrypt

Return:

- 0 On successfully decrypting message.
- BAD_ALIGN_E may be returned on block align error.
- BAD_LENGTH_E will be returned if the input length isn't a multiple of the AES block length, when the library is built with WOLFSSL_AES_CBC_LENGTH_CHECKS.

Example

```
Aes dec;
int ret = 0;
// initialize dec with wc_AesInit and wc_AesSetKey, using direction
// AES_DECRYPTION
byte cipher[AES_BLOCK_SIZE * n]; // some multiple of 16 bytes
// fill cipher with cipher text
byte plain [AES_BLOCK_SIZE * n];
```

```
if ((ret = wc_AesCbcDecrypt(&dec, plain, cipher, sizeof(cipher))) != 0 ) {
// block align error
}
```

C.1.2.5 function wc_AesCtrEncrypt

```
int wc_AesCtrEncrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

Encrypts/Decrypts a message from the input buffer in, and places the resulting cipher text in the output buffer out using CTR mode with AES. This function is only enabled if WOLFSSL_AES_COUNTER is enabled at compile time. The AES structure should be initialized through AesSetKey before calling this function. Note that this function is used for both decryption and encryption. *NOTE:* Regarding using same API for encryption and decryption. User should differentiate between Aes structures for encrypt/decrypt.

Parameters:

- **aes** pointer to the AES object used to decrypt data
- **out** pointer to the output buffer in which to store the cipher text of the encrypted message size must be a multiple of AES_BLOCK_LENGTH, padded if necessary
- **in** pointer to the input buffer containing plain text to be encrypted size must be a multiple of AES_BLOCK_LENGTH, padded if necessary
- **sz** size of the input plain text

See: wc_AesSetKey

Return: int integer values corresponding to wolfSSL error or success status

Example

```
Aes enc;
Aes dec;
// initialize enc and dec with wc_AesInit and wc_AesSetKeyDirect, using
// direction AES_ENCRYPTION since the underlying API only calls Encrypt
// and by default calling encrypt on a cipher results in a decryption of
// the cipher

byte msg[AES_BLOCK_SIZE * n]; //n being a positive integer making msg
some multiple of 16 bytes
// fill plain with message text
byte cipher[AES_BLOCK_SIZE * n];
byte decrypted[AES_BLOCK_SIZE * n];
wc_AesCtrEncrypt(&enc, cipher, msg, sizeof(msg)); // encrypt plain
wc_AesCtrEncrypt(&dec, decrypted, cipher, sizeof(cipher));
// decrypt cipher text
```

C.1.2.6 function wc_AesEncryptDirect

```
int wc_AesEncryptDirect(
    Aes * aes,
    byte * out,
    const byte * in
)
```

This function is a one-block encrypt of the input block, in, into the output block, out. It uses the key of the provided AES structure, which should be initialized with wc_AesSetKey before calling this function. wc_AesSetKey should have been called with the iv set to NULL. This is only enabled if the configure option WOLFSSL_AES_DIRECT is enabled. **Warning:** In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB API's directly whenever possible. When WC_AES_DIRECT_CRYPTO_CB is defined (set automatically for WOLF_CRYPTO_CB builds that use the generic AES Direct code, not a port's own version), if the AES object has a crypto callback device (devId), that device is asked first to encrypt the block as a one-block ECB request. The software key is used only if the device declines, and a device error is returned as-is. A device that owns the key must handle the request or return an error; it must not return CRYPTOCB_UNAVAILABLE. With WOLF_CRYPTO_CB_FIND, the device find callback picks the device, so a device can be asked even when the AES object has no devId.

Parameters:

- **aes** pointer to the AES object used to encrypt data
- **out** pointer to the output buffer in which to store the cipher text of the encrypted message
- **in** pointer to the input buffer containing plain text to be encrypted

See:

- wc_AesDecryptDirect
- wc_AesSetKeyDirect

Return: int integer values corresponding to wolfSSL error or success status

Example

```
Aes enc;
// initialize enc with wc_AesInit and wc_AesSetKey, using direction
// AES_ENCRYPTION
byte msg [AES_BLOCK_SIZE]; // 16 bytes
// initialize msg with plain text to encrypt
byte cipher[AES_BLOCK_SIZE];
wc_AesEncryptDirect(&enc, cipher, msg);
```

C.1.2.7 function wc_AesDecryptDirect

```
int wc_AesDecryptDirect(
    Aes * aes,
    byte * out,
    const byte * in
)
```

This function is a one-block decrypt of the input block, in, into the output block, out. It uses the key of the provided AES structure, which should be initialized with wc_AesSetKey before calling this function. wc_AesSetKey should have been called with the iv set to NULL. This is only enabled if the configure option WOLFSSL_AES_DIRECT is enabled. **Warning:** In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB API's directly whenever possible. When WC_AES_DIRECT_CRYPTO_CB is defined (set automatically for WOLF_CRYPTO_CB builds that use the generic AES Direct code, not a port's own version), if the AES object has a crypto callback device (devId), that device is asked first to decrypt the block as a one-block ECB request. The software key is used only if the device declines, and a device error is returned as-is. A device that owns the key must handle the request or return an error; it must not return CRYPTOCB_UNAVAILABLE. With WOLF_CRYPTO_CB_FIND, the device find callback picks the device, so a device can be asked even when the AES object has no devId.

Parameters:

- **aes** pointer to the AES object used to encrypt data
- **out** pointer to the output buffer in which to store the plain text of the decrypted cipher text
- **in** pointer to the input buffer containing cipher text to be decrypted

See:

- wc_AesEncryptDirect
- wc_AesSetKeyDirect

Return: int integer values corresponding to wolfSSL error or success status

Example

```
Aes dec;
// initialize enc with wc_AesInit and wc_AesSetKey, using direction
// AES_DECRYPTION
byte cipher [AES_BLOCK_SIZE]; // 16 bytes
// initialize cipher with cipher text to decrypt
byte msg[AES_BLOCK_SIZE];
wc_AesDecryptDirect(&dec, msg, cipher);
```

C.1.2.8 function wc_AesSetKeyDirect

```
int wc_AesSetKeyDirect(
    Aes * aes,
    const byte * key,
    word32 len,
    const byte * iv,
    int dir
)
```

This function is used to set the AES keys for CTR mode with AES. It initializes an AES object with the given key, iv (initialization vector), and encryption dir (direction). It is only enabled if the configure option WOLFSSL_AES_DIRECT is enabled. Currently wc_AesSetKeyDirect uses wc_AesSetKey internally. **Warning:** In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB API's directly whenever possible.

Parameters:

- **aes** pointer to the AES object used to encrypt data
- **key** 16, 24, or 32 byte secret key for encryption and decryption
- **len** length of the key passed in
- **iv** initialization vector used to initialize the key
- **dir** Cipher direction. Set AES_ENCRYPTION to encrypt, or AES_DECRYPTION to decrypt. (See enum in wolfssl/wolfcrypt/aes.h) (NOTE: If using wc_AesSetKeyDirect with Aes Counter mode (Stream cipher) only use AES_ENCRYPTION for both encrypting and decrypting)

See:

- wc_AesEncryptDirect
- wc_AesDecryptDirect
- wc_AesSetKey

Return:

- 0 On successfully setting the key.
- BAD_FUNC_ARG Returned if the given key is an invalid length.

Example

```
Aes enc;
int ret = 0;
byte key[] = { some 16, 24, or 32 byte key };
byte iv[]  = { some 16 byte iv };

if (ret = wc_AesInit(&enc, HEAP_HINT, INVALID_DEVID) != 0) {
    // failed to initialize aes key
}
if (ret = wc_AesSetKeyDirect(&enc, key, sizeof(key), iv,
AES_ENCRYPTION) != 0) {
// failed to set aes key
}
```

C.1.2.9 function wc_AesGcmSetKey

```
int wc_AesGcmSetKey(
    Aes * aes,
    const byte * key,
    word32 len
)
```

This function is used to set the key for AES GCM (Galois/Counter Mode). It initializes an AES object with the given key. It is only enabled if the configure option HAVE_AESGCM is enabled at compile time.

Parameters:

- **aes** pointer to the AES object used to encrypt data
- **key** 16, 24, or 32 byte secret key for encryption and decryption
- **len** length of the key passed in

See:

- wc_AesGcmEncrypt
- wc_AesGcmDecrypt

Return:

- 0 On successfully setting the key.
- BAD_FUNC_ARG Returned if the given key is an invalid length.

Example

```
Aes enc;
int ret = 0;
byte key[] = { some 16, 24,32 byte key };
if (ret = wc_AesInit(&enc, HEAP_HINT, INVALID_DEVID) != 0) {
    // failed to initialize aes key
}
if (ret = wc_AesGcmSetKey(&enc, key, sizeof(key)) != 0) {
// failed to set aes key
}
```

C.1.2.10 function wc_AesSetTagLen

```
int wc_AesSetTagLen(
    Aes * aes,
    word32 tagLen
)
```

This function associates one tag length with the key held by an AES object, as SP 800-38D section 5.2.1.2 and SP 800-38C section 5.3 require. Once set, AES-GCM and AES-CCM calls with that key accept only that tag length. wc_AesGcmSetKey() and wc_AesCcmSetKey() clear the association. The mode still applies its own list of allowed tag lengths on top of this.

Parameters:

- **aes** pointer to the AES object holding the key
- **tagLen** tag length to associate with the key

See:

- wc_AesGcmSetKey
- wc_AesCcmSetKey
- wc_CmacSetTagLen

Return:

- 0 On success.
- BAD_FUNC_ARG Returned if aes is NULL, the length is zero or larger than the AES block size, or a different length is already associated.

WOLFSSL_NO_AES_TAG_ASSOCIATION builds all of this out, which departs from the requirement above, so a FIPS v7 or later build refuses it.

The first use of a key writes the length into the AES object, so a keyed object shared read only between threads for one-shot AES-GCM or AES-CCM needs its length set here before it is handed out.

The first use of a key fixes its tag length even without this call. Once a length is associated this call can repeat it but not change or clear it; set the key again through one of those two to start over. wc_AesSetKey() clears it as well on builds that use the software key schedule, but a backend with its own key setter does not. Hardware offload ports do not apply this association, and such a build would be validated as a hybrid module.

A crypto callback is checked here before the work is handed over, because the key it forwards is the one this object holds. A callback that builds a fresh context per call cannot see the association, so this is the only place it can be caught. A device key installed by id or label is the exception, it is never seen here, so its first tag length stays associated until the object is freed.

An OpenSSL-compat EVP context re-initialized without a new key keeps the association, so a later verify asking for a different tag length fails. That is the one key one tag length rule, not a caller error. EVP encrypt always tags at the full length and shortens afterwards, so it never reaches this check.

Example

```
Aes enc;
byte key[] = { some 16, 24, 32 byte key };
if (wc_AesGcmSetKey(&enc, key, sizeof(key)) != 0) {
    // failed to set aes key
}
if (wc_AesSetTagLen(&enc, 16) != 0) {
    // failed to associate the tag length
}
```

C.1.2.11 function wc_AesGcmEncrypt

```
int wc_AesGcmEncrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz,
```

```
    const byte * iv,
    word32 ivSz,
    byte * authTag,
    word32 authTagSz,
    const byte * authIn,
    word32 authInSz
)
```

This function encrypts the input message, held in the buffer in, and stores the resulting cipher text in the output buffer out. It requires a new iv (initialization vector) for each call to encrypt. It also encodes the input authentication vector, authIn, into the authentication tag, authTag.

Parameters:

- **aes** - pointer to the AES object used to encrypt data
- **out** pointer to the output buffer in which to store the cipher text size must match in's size (sz)
- **in** pointer to the input buffer holding the message to encrypt size must be a multiple of AES_BLOCK_LENGTH, padded if necessary
- **sz** length of the input message to encrypt
- **iv** pointer to the buffer containing the initialization vector
- **ivSz** length of the initialization vector
- **authTag** pointer to the buffer in which to store the authentication tag
- **authTagSz** length of the desired authentication tag
- **authIn** pointer to the buffer containing the input authentication vector
- **authInSz** length of the input authentication vector

See:

- wc_AesGcmSetKey
- wc_AesGcmDecrypt

Return: 0 On successfully encrypting the input message

Note: When built with WOLFSSL_AFALG_XILINX_AES, the Xilinx AF_ALG kernel interface operates on a combined cipher text + tag buffer, so this function does not honor the exact-size buffer contract described below. Both in and out must be allocated with WC_AES_BLOCK_SIZE (16) bytes of room beyond sz: sz + 16 bytes are sent to the kernel from in (the trailing 16 bytes are scratch space for the tag and their contents are irrelevant), and sz + 16 bytes are read back into out. The tag is additionally copied out to authTag as usual. Both buffers should also be aligned to WOLFSSL_XILINX_ALIGN; an unaligned in is staged through a temporary allocation, or rejected with BAD_ALIGN_E if NO_WOLFSSL_ALLOC_ALIGN is defined.

Example

```
Aes enc;
// initialize Aes structure by calling wc_AesInit() and wc_AesGcmSetKey

byte plain[AES_BLOCK_LENGTH * n]; //n being a positive integer
making plain some multiple of 16 bytes
// initialize plain with msg to encrypt
byte cipher[sizeof(plain)];
byte iv[] = // some 16 byte iv
byte authTag[AUTH_TAG_LENGTH];
byte authIn[] = // Authentication Vector

wc_AesGcmEncrypt(&enc, cipher, plain, sizeof(cipher), iv, sizeof(iv),
        authTag, sizeof(authTag), authIn, sizeof(authIn));
```

C.1.2.12 function wc_AesGcmDecrypt

```
int wc_AesGcmDecrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    const byte * iv,
    word32 ivSz,
    const byte * authTag,
    word32 authTagSz,
    const byte * authIn,
    word32 authInSz
)
```

This function decrypts the input cipher text, held in the buffer in, and stores the resulting message text in the output buffer out. It also checks the input authentication vector, authIn, against the supplied authentication tag, authTag. If a nonzero error code is returned, the output data is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.

Parameters:

- **aes** pointer to the AES object used to encrypt data
- **out** pointer to the output buffer in which to store the message text size must match in's size (sz)
- **in** pointer to the input buffer holding the cipher text to decrypt size must be a multiple of AES_BLOCK_LENGTH, padded if necessary
- **sz** length of the cipher text to decrypt
- **iv** pointer to the buffer containing the initialization vector
- **ivSz** length of the initialization vector
- **authTag** pointer to the buffer containing the authentication tag
- **authTagSz** length of the desired authentication tag
- **authIn** pointer to the buffer containing the input authentication vector
- **authInSz** length of the input authentication vector

See:

- wc_AesGcmSetKey
- wc_AesGcmEncrypt

Return:

- 0 On successfully decrypting and authenticating the input message
- AES_GCM_AUTH_E If the authentication tag does not match the supplied authentication code vector, authTag.

Note: When built with WOLFSSL_AFALG_XILINX_AES, the Xilinx AF_ALG kernel interface operates on a combined cipher text + tag buffer, so this function does not honor the exact-size buffer contract described below. Both in and out must be allocated with WC_AES_BLOCK_SIZE (16) bytes of room beyond sz. The tag to check against is written into in + sz by this function, which means the in buffer is modified even though it is declared const, and sz + 16 bytes are read back into out. Both buffers should also be aligned to WOLFSSL_XILINX_ALIGN; an unaligned in is staged through a temporary allocation, or rejected with BAD_ALIGN_E if NO_WOLFSSL_ALLOC_ALIGN is defined.

Example

```
Aes enc; //can use the same struct as was passed to wc_AesGcmEncrypt
// initialize aes structure by calling wc_AesInit and wc_AesGcmSetKey
// if not already done
```

```
byte cipher[AES_BLOCK_LENGTH * n]; //n being a positive integer
making cipher some multiple of 16 bytes
// initialize cipher with cipher text to decrypt
byte output[sizeof(cipher)];
byte iv[] = // some 16 byte iv
byte authTag[AUTH_TAG_LENGTH];
byte authIn[] = // Authentication Vector

wc_AesGcmDecrypt(&enc, output, cipher, sizeof(cipher), iv, sizeof(iv),
        authTag, sizeof(authTag), authIn, sizeof(authIn));
```

C.1.2.13 function wc_GmacSetKey

```
int wc_GmacSetKey(
    Gmac * gmac,
    const byte * key,
    word32 len
)
```

This function initializes and sets the key for a GMAC object to be used for Galois Message Authentication.

Parameters:

- **gmac** pointer to the gmac object used for authentication
- **key** 16, 24, or 32 byte secret key for authentication
- **len** length of the key

See:

- wc_GmacUpdate
- wc_AesInit

Return:

- 0 On successfully setting the key
- BAD_FUNC_ARG Returned if key length is invalid.

Example

```
Gmac gmac;
key[] = { some 16, 24, or 32 byte length key };
wc_AesInit(gmac.aes, HEAP_HINT, INVALID_DEVID); // Make sure devId updated
wc_GmacSetKey(&gmac, key, sizeof(key));
```

C.1.2.14 function wc_GmacUpdate

```
int wc_GmacUpdate(
    Gmac * gmac,
    const byte * iv,
    word32 ivSz,
    const byte * authIn,
    word32 authInSz,
    byte * authTag,
    word32 authTagSz
)
```

This function generates the Gmac hash of the authIn input and stores the result in the authTag buffer. After running wc_GmacUpdate, one should compare the generated authTag to a known authentication tag to verify the authenticity of a message.

Parameters:

- **gmac** pointer to the gmac object used for authentication
- **iv** initialization vector used for the hash
- **ivSz** size of the initialization vector used
- **authIn** pointer to the buffer containing the authentication vector to verify
- **authInSz** size of the authentication vector
- **authTag** pointer to the output buffer in which to store the Gmac hash
- **authTagSz** the size of the output buffer used to store the Gmac hash

See:

- wc_GmacSetKey
- wc_AesInit

Return: 0 On successfully computing the Gmac hash.

Example

```
Gmac gmac;
key[] = { some 16, 24, or 32 byte length key };
iv[] = { some 16 byte length iv };

wc_AesInit(gmac.aes, HEAP_HINT, INVALID_DEVID); // Make sure devId updated
wc_GmacSetKey(&gmac, key, sizeof(key));
authIn[] = { some 16 byte authentication input };
tag[AES_BLOCK_SIZE]; // will store authentication code

wc_GmacUpdate(&gmac, iv, sizeof(iv), authIn, sizeof(authIn), tag,
sizeof(tag));
```

C.1.2.15 function wc_AesCcmSetKey

```
int wc_AesCcmSetKey(
    Aes * aes,
    const byte * key,
    word32 keySz
)
```

This function sets the key for an AES object using CCM (Counter with CBC-MAC). It takes a pointer to an AES structure and initializes it with supplied key.

Parameters:

- **aes** aes structure in which to store the supplied key
- **key** 16, 24, or 32 byte secret key for encryption and decryption
- **keySz** size of the supplied key

See:

- wc_AesCcmEncrypt
- wc_AesCcmDecrypt

Return: none

Example

```
Aes enc;
key[] = { some 16, 24, or 32 byte length key };

wc_AesInit(&enc, HEAP_HINT, INVALID_DEVID); // Make sure devId updated
wc_AesCcmSetKey(&enc, key, sizeof(key));
```

C.1.2.16 function wc_AesCcmEncrypt

```
int wc_AesCcmEncrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * nonce,
    word32 nonceSz,
    byte * authTag,
    word32 authTagSz,
    const byte * authIn,
    word32 authInSz
)
```

This function encrypts the input message, in, into the output buffer, out, using CCM (Counter with CBC-MAC). It subsequently calculates and stores the authorization tag, authTag, from the authIn input.

Parameters:

- **aes** pointer to the AES object used to encrypt data
- **out** pointer to the output buffer in which to store the cipher text
- **in** pointer to the input buffer holding the message to encrypt
- **sz** length of the input message to encrypt
- **nonce** pointer to the buffer containing the nonce (number only used once)
- **nonceSz** length of the nonce
- **authTag** pointer to the buffer in which to store the authentication tag
- **authTagSz** length of the desired authentication tag
- **authIn** pointer to the buffer containing the input authentication vector
- **authInSz** length of the input authentication vector

See:

- wc_AesCcmSetKey
- wc_AesCcmDecrypt

Return: none

Example

```
Aes enc;
// initialize enc with wc_AesInit and wc_AesCcmSetKey

nonce[] = { initialize nonce };
plain[] = { some plain text message };
cipher[sizeof(plain)];

authIn[] = { some 16 byte authentication input };
tag[AES_BLOCK_SIZE]; // will store authentication code

wc_AesCcmEncrypt(&enc, cipher, plain, sizeof(plain), nonce, sizeof(nonce),
        tag, sizeof(tag), authIn, sizeof(authIn));
```

C.1.2.17 function wc_AesCcmDecrypt

```
int wc_AesCcmDecrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * nonce,
    word32 nonceSz,
    const byte * authTag,
    word32 authTagSz,
    const byte * authIn,
    word32 authInSz
)
```

This function decrypts the input cipher text, in, into the output buffer, out, using CCM (Counter with CBC-MAC). It subsequently calculates the authorization tag, authTag, from the authIn input. If a nonzero error code is returned, the output data is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.

Parameters:

- **aes** pointer to the AES object used to encrypt data
- **out** pointer to the output buffer in which to store the cipher text
- **in** pointer to the input buffer holding the message to encrypt
- **sz** length of the input cipher text to decrypt
- **nonce** pointer to the buffer containing the nonce (number only used once)
- **nonceSz** length of the nonce
- **authTag** pointer to the buffer in which to store the authentication tag
- **authTagSz** length of the desired authentication tag
- **authIn** pointer to the buffer containing the input authentication vector
- **authInSz** length of the input authentication vector

See:

- wc_AesCcmSetKey
- wc_AesCcmEncrypt

Return:

- 0 On successfully decrypting the input message
- AES_CCM_AUTH_E If the authentication tag does not match the supplied authentication code vector, authTag.

Example

```
Aes dec;
// initialize dec with wc_AesInit and wc_AesCcmSetKey

nonce[] = { initialize nonce };
cipher[] = { encrypted message };
plain[sizeof(cipher)];

authIn[] = { some 16 byte authentication input };
tag[AES_BLOCK_SIZE] = { authentication tag received for verification };

int return = wc_AesCcmDecrypt(&dec, plain, cipher, sizeof(cipher),
nonce, sizeof(nonce),tag, sizeof(tag), authIn, sizeof(authIn));
if(return != 0) {
```

```
// decrypt error, invalid authentication code
}
```

C.1.2.18 function wc_AesXtsInit

```
int wc_AesXtsInit(
    XtsAes * aes,
    void * heap,
    int devId
)
```

This is to initialize an AES-XTS context. It is up to user to call wc_AesXtsFree on aes key when done.

Parameters:

- **aes** AES keys for encrypt/decrypt process
- **heap** heap hint to use for memory. Can be NULL
- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID (-2) if not used

See:

- wc_AesXtsSetKey
- wc_AesXtsSetKeyNoInit
- wc_AesXtsEncrypt
- wc_AesXtsDecrypt
- wc_AesXtsFree

Return: 0 Success

Example

```
XtsAes aes;

if(wc_AesXtsInit(&aes, NULL, INVALID_DEVID) != 0)
{
    // Handle error
}
if(wc_AesXtsSetKeyNoInit(&aes, key, sizeof(key), AES_ENCRYPTION) != 0)
{
    // Handle error
}
wc_AesXtsFree(&aes);
```

C.1.2.19 function wc_AesXtsSetKeyNoInit

```
int wc_AesXtsSetKeyNoInit(
    XtsAes * aes,
    const byte * key,
    word32 len,
    int dir
)
```

This is to help with setting keys to correct encrypt or decrypt type, after first calling wc_AesXtsInit(). It is up to user to call wc_AesXtsFree on aes key when done.

Parameters:

- **aes** AES keys for encrypt/decrypt process
- **key** buffer holding aes key | tweak key
- **len** length of key buffer in bytes. Should be twice that of key size. i.e. 32 for a 16 byte key.

- **dir** direction, either AES_ENCRYPTION or AES_DECRYPTION

See:

- wc_AesXtsEncrypt
- wc_AesXtsDecrypt
- wc_AesXtsFree

Return: 0 Success

Example

```
XtsAes aes;

if(wc_AesXtsInit(&aes, NULL, 0) != 0)
{
    // Handle error
}
if(wc_AesXtsSetKeyNoInit(&aes, key, sizeof(key), AES_ENCRYPTION, NULL, 0)
   != 0)
{
    // Handle error
}
wc_AesXtsFree(&aes);
```

C.1.2.20 function wc_AesXtsSetKey

```
int wc_AesXtsSetKey(
    XtsAes * aes,
    const byte * key,
    word32 len,
    int dir,
    void * heap,
    int devId
)
```

This is to help with setting keys to correct encrypt or decrypt type. It is up to user to call wc_AesXtsFree on aes key when done.

Parameters:

- **aes** AES keys for encrypt/decrypt process
- **key** buffer holding aes key | tweak key
- **len** length of key buffer in bytes. Should be twice that of key size. i.e. 32 for a 16 byte key.
- **dir** direction, either AES_ENCRYPTION or AES_DECRYPTION
- **heap** heap hint to use for memory. Can be NULL
- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID (-2) if not used

See:

- wc_AesXtsInit
- wc_AesXtsSetKeyNoInit
- wc_AesXtsEncrypt
- wc_AesXtsDecrypt
- wc_AesXtsFree

Return: 0 Success

Example

```
XtsAes aes;

if(wc_AesXtsSetKey(&aes, key, sizeof(key), AES_ENCRYPTION, NULL, INVALID_DEVID)
 ↪  != 0)
{
    // Handle error
}
wc_AesXtsFree(&aes);
```

C.1.2.21 function wc_AesXtsEncryptSector

```
int wc_AesXtsEncryptSector(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    word64 sector
)
```

Same process as wc_AesXtsEncrypt but uses a word64 type as the tweak value instead of a byte array. This just converts the word64 to a byte array and calls wc_AesXtsEncrypt.

Parameters:

- **aes** AES keys to use for block encrypt/decrypt
- **out** output buffer to hold cipher text
- **in** input plain text buffer to encrypt
- **sz** size of both out and in buffers
- **sector** value to use for tweak

See:

- wc_AesXtsEncrypt
- wc_AesXtsDecrypt
- wc_AesXtsInit
- wc_AesXtsSetKeyNoInit
- wc_AesXtsSetKey
- wc_AesXtsFree

Return: 0 Success

Example

```
XtsAes aes;
unsigned char plain[SIZE];
unsigned char cipher[SIZE];
word64 s = VALUE;

//set up keys with AES_ENCRYPTION as dir

if(wc_AesXtsEncryptSector(&aes, cipher, plain, SIZE, s) != 0)
{
    // Handle error
}
wc_AesXtsFree(&aes);
```

C.1.2.22 function wc_AesXtsDecryptSector

```
int wc_AesXtsDecryptSector(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    word64 sector
)
```

Same process as wc_AesXtsDecrypt but uses a word64 type as the tweak value instead of a byte array. This just converts the word64 to a byte array.

Parameters:

- **aes** AES keys to use for block encrypt/decrypt
- **out** output buffer to hold plain text
- **in** input cipher text buffer to decrypt
- **sz** size of both out and in buffers
- **sector** value to use for tweak

See:

- wc_AesXtsEncrypt
- wc_AesXtsDecrypt
- wc_AesXtsInit
- wc_AesXtsSetKeyNoInit
- wc_AesXtsSetKey
- wc_AesXtsFree

Return: 0 Success

Example

```
XtsAes aes;
unsigned char plain[SIZE];
unsigned char cipher[SIZE];
word64 s = VALUE;

//set up aes key with AES_DECRYPTION as dir and tweak with AES_ENCRYPTION

if(wc_AesXtsDecryptSector(&aes, plain, cipher, SIZE, s) != 0)
{
    // Handle error
}
wc_AesXtsFree(&aes);
```

C.1.2.23 function wc_AesXtsEncrypt

```
int wc_AesXtsEncrypt(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    const byte * i,
    word32 iSz
)
```

AES with XTS mode. (XTS) XEX encryption with Tweak and cipher text Stealing.

Parameters:

- **aes** AES keys to use for block encrypt/decrypt
- **out** output buffer to hold cipher text
- **in** input plain text buffer to encrypt
- **sz** size of both out and in buffers
- **i** value to use for tweak
- **iSz** size of i buffer, should always be AES_BLOCK_SIZE but having this input adds a sanity check on how the user calls the function.

See:

- wc_AesXtsDecrypt
- wc_AesXtsInit
- wc_AesXtsSetKeyNoInit
- wc_AesXtsSetKey
- wc_AesXtsFree

Return: 0 Success

Example

```
XtsAes aes;
unsigned char plain[SIZE];
unsigned char cipher[SIZE];
unsigned char i[AES_BLOCK_SIZE];

//set up key with AES_ENCRYPTION as dir

if(wc_AesXtsEncrypt(&aes, cipher, plain, SIZE, i, sizeof(i)) != 0)
{
    // Handle error
}
wc_AesXtsFree(&aes);
```

C.1.2.24 function wc_AesXtsDecrypt

```
int wc_AesXtsDecrypt(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    const byte * i,
    word32 iSz
)
```

Same process as encryption but Aes key is AES_DECRYPTION type.

Parameters:

- **aes** AES keys to use for block encrypt/decrypt
- **out** output buffer to hold plain text
- **in** input cipher text buffer to decrypt
- **sz** size of both out and in buffers
- **i** value to use for tweak
- **iSz** size of i buffer, should always be AES_BLOCK_SIZE but having this input adds a sanity check on how the user calls the function.

See:

- wc_AesXtsEncrypt

- wc_AesXtsInit
- wc_AesXtsSetKeyNoInit
- wc_AesXtsSetKey
- wc_AesXtsFree

Return: 0 Success

Example

```
XtsAes aes;
unsigned char plain[SIZE];
unsigned char cipher[SIZE];
unsigned char i[AES_BLOCK_SIZE];

//set up key with AES_DECRYPTION as dir and tweak with AES_ENCRYPTION

if(wc_AesXtsDecrypt(&aes, plain, cipher, SIZE, i, sizeof(i)) != 0)
{
    // Handle error
}
wc_AesXtsFree(&aes);
```

C.1.2.25 function wc_AesXtsFree

```
int wc_AesXtsFree(
    XtsAes * aes
)
```

This is to free up any resources used by the XtsAes structure.

Parameters:

- **aes** AES keys to free

See:

- wc_AesXtsEncrypt
- wc_AesXtsDecrypt
- wc_AesXtsInit
- wc_AesXtsSetKeyNoInit
- wc_AesXtsSetKey

Return: 0 Success

Example

```
XtsAes aes;

if(wc_AesXtsSetKey(&aes, key, sizeof(key), AES_ENCRYPTION, NULL, 0) != 0)
{
    // Handle error
}
wc_AesXtsFree(&aes);
```

C.1.2.26 function wc_AesInit

```
int wc_AesInit(
    Aes * aes,
    void * heap,
```

```
    int devId
)
```

Initialize Aes structure. Sets heap hint to be used and ID for use with async hardware. It is up to the user to call wc_AesFree on the Aes structure when done.

Parameters:

- **aes** aes structure in to initialize
- **heap** heap hint to use for malloc / free if needed
- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID (-2) if not used

See:

- wc_AesSetKey
- wc_AesSetIV
- wc_AesFree

Return: 0 Success

Example

```
Aes enc;
void* hint = NULL;
int devId = INVALID_DEVID; //if not using async INVALID_DEVID is default

//heap hint could be set here if used

wc_AesInit(&enc, hint, devId);
```

C.1.2.27 function wc_AesFree

```
void wc_AesFree(
    Aes * aes
)
```

free resources associated with the Aes structure when applicable. Internally may sometimes be a no-op but still recommended to call in all cases as a general best-practice (IE if application code is ported for use on new environments where the call is applicable).

Parameters:

- **aes** aes structure in to free

See: wc_AesInit

Return: no return (void function)

Example

```
Aes enc;
void* hint = NULL;
int devId = INVALID_DEVID; //if not using async INVALID_DEVID is default

//heap hint could be set here if used

wc_AesInit(&enc, hint, devId);
// ... do some interesting things ...
wc_AesFree(&enc);
```

C.1.2.28 function wc_AesCfbEncrypt

```
int wc_AesCfbEncrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

AES with CFB mode.

Parameters:

- **aes** AES keys to use for block encrypt/decrypt
- **out** output buffer to hold cipher text must be at least as large as inputbuffer)
- **in** input plain text buffer to encrypt
- **sz** size of input buffer

See:

- wc_AesCfbDecrypt
- wc_AesSetKey

Return: 0 Success and negative error values on failure

Example

```
Aes aes;
unsigned char plain[SIZE];
unsigned char cipher[SIZE];

//set up key with AES_ENCRYPTION as dir for both encrypt and decrypt

if(wc_AesCfbEncrypt(&aes, cipher, plain, SIZE) != 0)
{
    // Handle error
}
```

C.1.2.29 function wc_AesCfbDecrypt

```
int wc_AesCfbDecrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

AES with CFB mode.

Parameters:

- **aes** AES keys to use for block encrypt/decrypt
- **out** output buffer to hold decrypted text must be at least as large as inputbuffer)
- **in** input buffer to decrypt
- **sz** size of input buffer

See:

- wc_AesCfbEncrypt
- wc_AesSetKey

Return: 0 Success and negative error values on failure

Example

```
Aes aes;
unsigned char plain[SIZE];
unsigned char cipher[SIZE];

//set up key with AES_ENCRYPTION as dir for both encrypt and decrypt

if(wc_AesCfbDecrypt(&aes, plain, cipher, SIZE) != 0)
{
    // Handle error
}
```

C.1.2.30 function wc_AesSivEncrypt

```
int wc_AesSivEncrypt(
    const byte * key,
    word32 keySz,
    const byte * assoc,
    word32 assocSz,
    const byte * nonce,
    word32 nonceSz,
    const byte * in,
    word32 inSz,
    byte * siv,
    byte * out
)
```

This function performs SIV (synthetic initialization vector) encryption as described in RFC 5297.

Parameters:

- **key** Byte buffer containing the key to use.
- **keySz** Length of the key buffer in bytes.
- **assoc** Additional, authenticated associated data (AD).
- **assocSz** Length of AD buffer in bytes.
- **nonce** A number used once. Used by the algorithm in the same manner as the AD.
- **nonceSz** Length of nonce buffer in bytes.
- **in** Plaintext buffer to encrypt.
- **inSz** Length of plaintext buffer.
- **siv** The SIV output by S2V (see RFC 5297 2.4).
- **out** Buffer to hold the ciphertext. Should be the same length as the plaintext buffer.

See: wc_AesSivDecrypt

Return:

- 0 On successful encryption.
- BAD_FUNC_ARG If key, SIV, or output buffer are NULL. Also returned if the key size isn't 32, 48, or 64 bytes.
- Other Other negative error values returned if AES or CMAC operations fail.

Example

```
byte key[] = { some 32, 48, or 64 byte key };
byte assoc[] = {0x01, 0x2, 0x3};
byte nonce[] = {0x04, 0x5, 0x6};
```

```
byte plainText[] = {0xDE, 0xAD, 0xBE, 0xEF};
byte siv[AES_BLOCK_SIZE];
byte cipherText[sizeof(plainText)];
if (wc_AesSivEncrypt(key, sizeof(key), assoc, sizeof(assoc), nonce,
    sizeof(nonce), plainText, sizeof(plainText), siv, cipherText) != 0) {
    // failed to encrypt
}
```

C.1.2.31 function wc_AesSivDecrypt

```
int wc_AesSivDecrypt(
    const byte * key,
    word32 keySz,
    const byte * assoc,
    word32 assocSz,
    const byte * nonce,
    word32 nonceSz,
    const byte * in,
    word32 inSz,
    byte * siv,
    byte * out
)
```

This function performs SIV (synthetic initialization vector) decryption as described in RFC 5297. If a nonzero error code is returned, the output data is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.

Parameters:

- **key** Byte buffer containing the key to use.
- **keySz** Length of the key buffer in bytes.
- **assoc** Additional, authenticated associated data (AD).
- **assocSz** Length of AD buffer in bytes.
- **nonce** A number used once. Used by the underlying algorithm in the same manner as the AD.
- **nonceSz** Length of nonce buffer in bytes.
- **in** Ciphertext buffer to decrypt.
- **inSz** Length of ciphertext buffer.
- **siv** The SIV that accompanies the ciphertext (see RFC 5297 2.4).
- **out** Buffer to hold the decrypted plaintext. Should be the same length as the ciphertext buffer.

See: wc_AesSivEncrypt

Return:

- 0 On successful decryption.
- BAD_FUNC_ARG If key, SIV, or output buffer are NULL. Also returned if the key size isn't 32, 48, or 64 bytes.
- AES_SIV_AUTH_E If the SIV derived by S2V doesn't match the input SIV (see RFC 5297 2.7).
- Other Other negative error values returned if AES or CMAC operations fail.

Example

```
byte key[] = { some 32, 48, or 64 byte key };
byte assoc[] = {0x01, 0x2, 0x3};
byte nonce[] = {0x04, 0x5, 0x6};
byte cipherText[] = {0xDE, 0xAD, 0xBE, 0xEF};
byte siv[AES_BLOCK_SIZE] = { the SIV that came with the ciphertext };
byte plainText[sizeof(cipherText)];
```

```
if (wc_AesSivDecrypt(key, sizeof(key), assoc, sizeof(assoc), nonce,
    sizeof(nonce), cipherText, sizeof(cipherText), siv, plainText) != 0) {
    // failed to decrypt
}
```

C.1.2.32 function wc_AesEaxEncryptAuth

```
WOLFSSL_API int wc_AesEaxEncryptAuth(
    const byte * key,
    word32 keySz,
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * nonce,
    word32 nonceSz,
    byte * authTag,
    word32 authTagSz,
    const byte * authIn,
    word32 authInSz
)
```

This function performs AES EAX encryption and authentication as described in "EAX: A Conventional Authenticated-Encryption Mode" (https://eprint.iacr.org/2003/069). It is a "one-shot" API that performs all encryption and authentication operations in one function call.

Parameters:

- **key** buffer containing the key to use
- **keySz** length of the key buffer in bytes
- **out** buffer to hold the ciphertext. Should be the same length as the plaintext buffer
- **in** plaintext buffer to encrypt
- **inSz** length of plaintext buffer
- **nonce** the cryptographic nonce to use for EAX operations
- **nonceSz** length of nonce buffer in bytes
- **authTag** pointer to the buffer in which to store the authentication tag
- **authTagSz** length of the desired authentication tag
- **authIn** pointer to the buffer containing input data to authenticate
- **authInSz** length of the input authentication data

See: wc_AesEaxDecryptAuth

Return:

- 0 on successful encryption.
- BAD_FUNC_ARG if input or output buffers are NULL. Also returned if the key size isn't a valid AES key size (16, 24, or 32 bytes)
- other negative error values returned if AES or CMAC operations fail.

Example

```
byte key[] = { some 32, 48, or 64 byte key };
byte nonce[] = {0x04, 0x5, 0x6};
byte plainText[] = {0xDE, 0xAD, 0xBE, 0xEF};
byte authIn[] = {0x01, 0x2, 0x3};

byte cipherText[sizeof(plainText)]; // output ciphertext
byte authTag[length, up to AES_BLOCK_SIZE]; // output authTag
```

```
if (wc_AesEaxEncrypt(key, sizeof(key),
                     cipherText, plainText, sizeof(plainText),
                     nonce, sizeof(nonce),
                     authTag, sizeof(authTag),
                     authIn, sizeof(authIn)) != 0) {
    // failed to encrypt
}
```

C.1.2.33 function wc_AesEaxDecryptAuth

```
WOLFSSL_API int wc_AesEaxDecryptAuth(
    const byte * key,
    word32 keySz,
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * nonce,
    word32 nonceSz,
    const byte * authTag,
    word32 authTagSz,
    const byte * authIn,
    word32 authInSz
)
```

This function performs AES EAX decryption and authentication as described in "EAX: A Conventional Authenticated-Encryption Mode" (https://eprint.iacr.org/2003/069). It is a "one-shot" API that performs all decryption and authentication operations in one function call. If a nonzero error code is returned, the output data is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.

Parameters:

- **key** byte buffer containing the key to use
- **keySz** length of the key buffer in bytes
- **out** buffer to hold the plaintext. Should be the same length as the input ciphertext buffer
- **in** ciphertext buffer to decrypt
- **inSz** length of ciphertext buffer
- **nonce** the cryptographic nonce to use for EAX operations
- **nonceSz** length of nonce buffer in bytes
- **authTag** buffer that holds the authentication tag to check the authenticity of the data against
- **authTagSz** Length of the input authentication tag
- **authIn** pointer to the buffer containing input data to authenticate
- **authInSz** length of the input authentication data

See: wc_AesEaxEncryptAuth

Return:

- 0 on successful decryption
- BAD_FUNC_ARG if input or output buffers are NULL. Also returned if the key size isn't a valid AES key size (16, 24, or 32 bytes)
- AES_EAX_AUTH_E If the authentication tag does not match the supplied authentication code vector authTag
- other negative error values returned if AES or CMAC operations fail.

Example

```
byte key[] = { some 32, 48, or 64 byte key };
byte nonce[] = {0x04, 0x5, 0x6};
byte cipherText[] = {0xDE, 0xAD, 0xBE, 0xEF};
byte authIn[] = {0x01, 0x2, 0x3};

byte plainText[sizeof(cipherText)]; // output plaintext
byte authTag[length, up to AES_BLOCK_SIZE]; // output authTag

if (wc_AesEaxDecrypt(key, sizeof(key),
                     cipherText, plainText, sizeof(plainText),
                     nonce, sizeof(nonce),
                     authTag, sizeof(authTag),
                     authIn, sizeof(authIn)) != 0) {
    // failed to encrypt
}
```

C.1.2.34 function wc_AesEaxInit

```
WOLFSSL_API int wc_AesEaxInit(
    AesEax * eax,
    const byte * key,
    word32 keySz,
    const byte * nonce,
    word32 nonceSz,
    const byte * authIn,
    word32 authInSz
)
```

This function initializes an AesEax object for use in authenticated encryption or decryption. This function must be called on an AesEax object before using it with any of the AES EAX incremental API functions. It does not need to be called if using the one-shot EAX API functions. All AesEax instances initialized with this function need to be freed with a call to wc_AesEaxFree() when done using the instance.

Parameters:

- **eax** AES EAX structure holding the context of the AEAD operation
- **key** 16, 24, or 32 byte secret key for encryption and decryption
- **keySz** length of the supplied key in bytes
- **nonce** the cryptographic nonce to use for EAX operations
- **nonceSz** length of nonce buffer in bytes
- **authIn** (optional) input data to add to the authentication stream This argument should be NULL if not used
- **authInSz** size in bytes of the input authentication data

See:

- wc_AesEaxEncryptUpdate
- wc_AesEaxDecryptUpdate
- wc_AesEaxAuthDataUpdate
- wc_AesEaxEncryptFinal
- wc_AesEaxDecryptFinal
- wc_AesEaxFree

Return:

- 0 on success
- error code on failure

Example

```
AesEax eax;
key[]   = { some 16, 24, or 32 byte length key };
nonce[] = { some arbitrary length nonce };
authIn[] = { some data to add to the authentication stream };
plainText[] = {some plaintext data to encrypt};

cipherText[sizeof(plainText)]; // buffer to hold cipherText
authTag[length, up to AES_BLOCK_SIZE]; // buffer to hold computed auth data

AesEax eax;

if ((ret = wc_AesEaxInit(eax,
                         key, keySz,
                         nonce, nonceSz,
                         authIn, authInSz)) != 0) {
    goto cleanup;
}

// if we wanted to add more auth data, we could provide it at this point,
// otherwise we use NULL for the authIn parameter, with authIn size of 0
if ((ret = wc_AesEaxEncryptUpdate(eax,
                                  cipherText, plainText, sizeof(plainText),
                                  NULL, 0)) != 0) {
    goto cleanup;
}

if ((ret = wc_AesEaxEncryptFinal(eax, authTag, sizeof(authTag))) != 0) {
    goto cleanup;
}

cleanup:
    wc_AesEaxFree(eax);
```

C.1.2.35 function wc_AesEaxEncryptUpdate

```
WOLFSSL_API int wc_AesEaxEncryptUpdate(
    AesEax * eax,
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * authIn,
    word32 authInSz
)
```

This function uses AES EAX to encrypt input data, and optionally, add more input data to the authentication stream. eax must have been previously initialized with a call to wc_AesEaxInit.

Parameters:

- **eax** AES EAX structure holding the context of the AEAD operation
- **out** output buffer holding the ciphertext
- **in** input buffer holding the plaintext to encrypt
- **inSz** size in bytes of the input data buffer

- **authIn** (optional) input data to add to the authentication stream This argument should be NULL if not used
- **authInSz** size in bytes of the input authentication data

See:

- wc_AesEaxInit
- wc_AesEaxDecryptUpdate
- wc_AesEaxAuthDataUpdate
- wc_AesEaxEncryptFinal
- wc_AesEaxDecryptFinal
- wc_AesEaxFree

Return:

- 0 on success
- error code on failure

Example

```
AesEax eax;
key[]   = { some 16, 24, or 32 byte length key };
nonce[] = { some arbitrary length nonce };
authIn[] = { some data to add to the authentication stream };
plainText[] = {some plaintext data to encrypt};

cipherText[sizeof(plainText)]; // buffer to hold cipherText
authTag[length, up to AES_BLOCK_SIZE]; // buffer to hold computed auth data

AesEax eax;

if ((ret = wc_AesEaxInit(eax,
                         key, keySz,
                         nonce, nonceSz,
                         authIn, authInSz)) != 0) {
    goto cleanup;
}

// if we wanted to add more auth data, we could provide it at this point,
// otherwise we use NULL for the authIn parameter, with authInSz of 0
if ((ret = wc_AesEaxEncryptUpdate(eax,
                                  cipherText, plainText, sizeof(plainText),
                                  NULL, 0)) != 0) {
    goto cleanup;
}

if ((ret = wc_AesEaxEncryptFinal(eax, authTag, sizeof(authTag))) != 0) {
    goto cleanup;
}

cleanup:
    wc_AesEaxFree(eax);
```

C.1.2.36 function wc_AesEaxDecryptUpdate

```
WOLFSSL_API int wc_AesEaxDecryptUpdate(
    AesEax * eax,
```

```
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * authIn,
    word32 authInSz
)
```

This function uses AES EAX to decrypt input data, and optionally, add more input data to the authentication stream. eax must have been previously initialized with a call to wc_AesEaxInit.

Parameters:

- **eax** AES EAX structure holding the context of the AEAD operation
- **out** output buffer holding the decrypted plaintext
- **in** input buffer holding the ciphertext
- **inSz** size in bytes of the input data buffer
- **authIn** (optional) input data to add to the authentication stream This argument should be NULL if not used
- **authInSz** size in bytes of the input authentication data

See:

- wc_AesEaxInit
- wc_AesEaxEncryptUpdate
- wc_AesEaxAuthDataUpdate
- wc_AesEaxEncryptFinal
- wc_AesEaxDecryptFinal
- wc_AesEaxFree

Return:

- 0 on success
- error code on failure

Example

```
AesEax eax;
key[]   = { some 16, 24, or 32 byte length key };
nonce[] = { some arbitrary length nonce };
authIn[] = { some data to add to the authentication stream };
cipherText[] = {some encrypted data};

plainText[sizeof(cipherText)]; // buffer to hold decrypted data
// auth tag is generated elsewhere by the encrypt AEAD operation
authTag[length, up to AES_BLOCK_SIZE] = { the auth tag };

AesEax eax;

if ((ret = wc_AesEaxInit(eax,
                         key, keySz,
                         nonce, nonceSz,
                         authIn, authInSz)) != 0) {
    goto cleanup;
}

// if we wanted to add more auth data, we could provide it at this point,
// otherwise we use NULL for the authIn parameter, with authInSz of 0
if ((ret = wc_AesEaxDecryptUpdate(eax,
                                  plainText, cipherText, sizeof(cipherText),
```

```
                                        NULL, 0)) != 0) {
    goto cleanup;
}

if ((ret = wc_AesEaxDecryptFinal(eax, authTag, sizeof(authTag))) != 0) {
    goto cleanup;
}

cleanup:
    wc_AesEaxFree(eax);
```

C.1.2.37 function wc_AesEaxAuthDataUpdate

```
WOLFSSL_API int wc_AesEaxAuthDataUpdate(
    AesEax * eax,
    const byte * authIn,
    word32 authInSz
)
```

This function adds input data to the authentication stream. eax must have been previously initialized with a call to wc_AesEaxInit.

Parameters:

- **eax** AES EAX structure holding the context of the AEAD operation
- **authIn** input data to add to the authentication stream
- **authInSz** size in bytes of the input authentication data

See:

- wc_AesEaxInit
- wc_AesEaxEncryptUpdate
- wc_AesEaxDecryptUpdate
- wc_AesEaxEncryptFinal
- wc_AesEaxDecryptFinal
- wc_AesEaxFree

Return:

- 0 on success
- error code on failure

Example

```
AesEax eax;
key[]   = { some 16, 24, or 32 byte length key };
nonce[] = { some arbitrary length nonce };
authIn[] = { some data to add to the authentication stream };
cipherText[] = {some encrypted data};

plainText[sizeof(cipherText)]; // buffer to hold decrypted data
// auth tag is generated elsewhere by the encrypt AEAD operation
authTag[length, up to AES_BLOCK_SIZE] = { the auth tag };

AesEax eax;

// No auth data to add here
if ((ret = wc_AesEaxInit(eax,
                         key, keySz,
```

```
                        nonce, nonceSz,
                        NULL, 0)) != 0) {
    goto cleanup;
}

// No auth data to add here, added later with wc_AesEaxAuthDataUpdate
if ((ret = wc_AesEaxDecryptUpdate(eax,
                                  plainText, cipherText, sizeof(cipherText),
                                  NULL, 0)) != 0) {
    goto cleanup;
}

if ((ret = wc_AesEaxAuthDataUpdate(eax, authIn, sizeof(authIn))) != 0) {
    goto cleanup;
}

if ((ret = wc_AesEaxDecryptFinal(eax, authTag, sizeof(authTag))) != 0) {
    goto cleanup;
}

cleanup:
    wc_AesEaxFree(eax);
```

C.1.2.38 function wc_AesEaxEncryptFinal

```
WOLFSSL_API int wc_AesEaxEncryptFinal(
    AesEax * eax,
    byte * authTag,
    word32 authTagSz
)
```

This function finalizes the encrypt AEAD operation, producing an auth tag over the current authentication stream. eax must have been previously initialized with a call to wc_AesEaxInit. When done using the AesEax context structure, make sure to free it using wc_AesEaxFree.

Parameters:

- **eax** AES EAX structure holding the context of the AEAD operation
- **authTag[out]** buffer that will hold the computed auth tag
- **authTagSz** size in bytes of authTag

See:

- wc_AesEaxInit
- wc_AesEaxEncryptUpdate
- wc_AesEaxDecryptUpdate
- wc_AesEaxAuthDataUpdate
- wc_AesEaxDecryptFinal
- wc_AesEaxFree

Return:

- 0 on success
- error code on failure

Example

```
AesEax eax;
key[]   = { some 16, 24, or 32 byte length key };
```

```
nonce[] = { some arbitrary length nonce };
authIn[] = { some data to add to the authentication stream };
plainText[] = {some plaintext data to encrypt};

cipherText[sizeof(plainText)]; // buffer to hold cipherText
authTag[length, up to AES_BLOCK_SIZE]; // buffer to hold computed auth data

AesEax eax;

if ((ret = wc_AesEaxInit(eax,
                         key, keySz,
                         nonce, nonceSz,
                         authIn, authInSz)) != 0) {
    goto cleanup;
}

// if we wanted to add more auth data, we could provide it at this point,
// otherwise we use NULL for the authIn parameter, with authInSz of 0
if ((ret = wc_AesEaxEncryptUpdate(eax,
                                  cipherText, plainText, sizeof(plainText),
                                  NULL, 0)) != 0) {
    goto cleanup;
}

if ((ret = wc_AesEaxEncryptFinal(eax, authTag, sizeof(authTag))) != 0) {
    goto cleanup;
}

cleanup:
    wc_AesEaxFree(eax);
```

C.1.2.39 function wc_AesEaxDecryptFinal

```
WOLFSSL_API int wc_AesEaxDecryptFinal(
    AesEax * eax,
    const byte * authIn,
    word32 authInSz
)
```

This function finalizes the decrypt AEAD operation, finalizing the auth tag computation and checking it for validity against the user supplied tag. eax must have been previously initialized with a call to wc_AesEaxInit. When done using the AesEax context structure, make sure to free it using wc_AesEaxFree.

Parameters:

- **eax** AES EAX structure holding the context of the AEAD operation
- **authIn** input auth tag to check computed auth tag against
- **authInSz** size in bytes of authIn

See:

- wc_AesEaxInit
- wc_AesEaxEncryptUpdate
- wc_AesEaxDecryptUpdate
- wc_AesEaxAuthDataUpdate
- wc_AesEaxEncryptFinal

- wc_AesEaxFree

Return:

- 0 if data is authenticated successfully
- AES_EAX_AUTH_E if the authentication tag does not match the supplied authentication code vector authIn
- other error code on failure

Example

```
AesEax eax;
key[]   = { some 16, 24, or 32 byte length key };
nonce[] = { some arbitrary length nonce };
authIn[] = { some data to add to the authentication stream };
cipherText[] = {some encrypted data};

plainText[sizeof(cipherText)]; // buffer to hold decrypted data
// auth tag is generated elsewhere by the encrypt AEAD operation
authTag[length, up to AES_BLOCK_SIZE] = { the auth tag };

AesEax eax;

if ((ret = wc_AesEaxInit(eax,
                         key, keySz,
                         nonce, nonceSz,
                         authIn, authInSz)) != 0) {
    goto cleanup;
}

// if we wanted to add more auth data, we could provide it at this point,
// otherwise we use NULL for the authIn parameter, with authInSz of 0
if ((ret = wc_AesEaxDecryptUpdate(eax,
                                  plainText, cipherText, sizeof(cipherText),
                                  NULL, 0)) != 0) {
    goto cleanup;
}

if ((ret = wc_AesEaxDecryptFinal(eax, authTag, sizeof(authTag))) != 0) {
    goto cleanup;
}

cleanup:
    wc_AesEaxFree(eax);
```

C.1.2.40 function wc_AesEaxFree

```
WOLFSSL_API int wc_AesEaxFree(
    AesEax * eax
)
```

This frees up any resources, specifically keys, used by the Aes instance inside the AesEax wrapper struct. It should be called on the AesEax struct after it has been initialized with wc_AesEaxInit, and all desired EAX operations are complete.

Parameters:

- **eaxAES** EAX instance to free

See:

- wc_AesEaxInit
- wc_AesEaxEncryptUpdate
- wc_AesEaxDecryptUpdate
- wc_AesEaxAuthDataUpdate
- wc_AesEaxEncryptFinal
- wc_AesEaxDecryptFinal

Return: 0 Success

Example

```
AesEax eax;

if(wc_AesEaxInit(eax, key, keySz, nonce, nonceSz, authIn, authInSz) != 0) {
    // handle errors, then free
    wc_AesEaxFree(&eax);
}
```

C.1.2.41 function wc_AesCtsEncrypt

```
int wc_AesCtsEncrypt(
    const byte * key,
    word32 keySz,
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * iv
)
```

This function performs AES encryption using Ciphertext Stealing (CTS) mode. It is a one-shot API that handles all operations in a single call.

Parameters:

- **key** pointer to the AES key used for encryption.
- **keySz** size of the AES key in bytes (16, 24, or 32 bytes).
- **out** buffer to hold the encrypted ciphertext. Must be at least the size of the input.
- **in** pointer to the plaintext input data to encrypt.
- **inSz** size of the plaintext input data in bytes.
- **iv** pointer to the initialization vector (IV) used for encryption. Must be 16 bytes.
- **key** pointer to the AES key used for encryption.
- **keySz** size of the AES key in bytes (16, 24, or 32 bytes).
- **out** buffer to hold the encrypted ciphertext. Must be at least the same size as the input plaintext.
- **in** pointer to the plaintext input data to encrypt.
- **inSz** size of the plaintext input data in bytes.
- **iv** pointer to the initialization vector (IV) used for encryption. Must be 16 bytes. *Example*

```
byte key[16] = { 0 };
byte iv[16] = { 0 };
byte plaintext[] = { 0x01, 0x02, 0x03, 0x04, 0x05 };
byte ciphertext[sizeof(plaintext)];
int ret = wc_AesCtsEncrypt(key, sizeof(key), ciphertext, plaintext,
                           sizeof(plaintext), iv);
if (ret != 0) {
    // handle encryption error
}
```

See:

- wc_AesCtsDecrypt
- wc_AesCtsDecrypt

Return:

- 0 on successful encryption.
- BAD_FUNC_ARG if input arguments are invalid.
- other negative error codes for encryption failures.
- 0 on successful encryption.
- BAD_FUNC_ARG if input arguments are invalid.
- other negative error codes for encryption failures.

Example

```
    byte key[16] = { 0 };
    byte iv[16] = { 0 };
    byte plaintext[] = { 0x01, 0x02, 0x03, 0x04, 0x05 };
    byte ciphertext[sizeof(plaintext)];

    int ret = wc_AesCtsEncrypt(key, sizeof(key), ciphertext, plaintext,
        sizeof(plaintext), iv);
    if (ret != 0) {
    // handle encryption error
}
```

C.1.2.42 function wc_AesCtsDecrypt

```
int wc_AesCtsDecrypt(
    const byte * key,
    word32 keySz,
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * iv
)
```

This function performs AES decryption using Ciphertext Stealing (CTS) mode. It is a one-shot API that handles all operations in a single call.

Parameters:

- **key** pointer to the AES key used for decryption.
- **keySz** size of the AES key in bytes (16, 24, or 32 bytes).
- **out** buffer to hold the decrypted plaintext. Must be at least the same size as the input ciphertext.
- **in** pointer to the ciphertext input data to decrypt.
- **inSz** size of the ciphertext input data in bytes.
- **iv** pointer to the initialization vector (IV) used for decryption. Must be 16 bytes. *Example*

```
byte key[16] = { 0 };
byte iv[16] = { 0 };
byte ciphertext[] = { 0x01, 0x02, 0x03, 0x04, 0x05 };
byte plaintext[sizeof(ciphertext)];
int ret = wc_AesCtsDecrypt(key, sizeof(key), plaintext, ciphertext,
                           sizeof(ciphertext), iv);
if (ret != 0) {
    // handle decryption error
}
```

See: wc_AesCtsEncrypt

Return:

- 0 on successful decryption.
- BAD_FUNC_ARG if input arguments are invalid.
- other negative error codes for decryption failures.

C.1.2.43 function wc_AesCtsEncryptUpdate

```
int wc_AesCtsEncryptUpdate(
    Aes * aes,
    byte * out,
    word32 * outSz,
    const byte * in,
    word32 inSz
)
```

This function performs an update step of the AES CTS encryption. It processes a chunk of plaintext and stores intermediate data.

Parameters:

- **aes** pointer to the Aes structure holding the context of the operation.
- **out** buffer to hold the encrypted ciphertext. Must be large enough to store the output from this update step.
- **outSz** size in bytes of the output data written to the out buffer. On input, it should contain the maximum number of bytes that can be written to the out buffer.
- **in** pointer to the plaintext input data to encrypt.
- **inSz** size of the plaintext input data in bytes. *Example*

```
Aes aes;
wc_AesInit(&aes, NULL, INVALID_DEVID);
byte key[16] = { 0 };
byte iv[16] = { 0 };
byte plaintext[] = { ... };
byte ciphertext[sizeof(plaintext)];
word32 outSz = sizeof(ciphertext);
wc_AesSetKey(&aes, key, sizeof(key), iv, AES_ENCRYPTION);
int ret = wc_AesCtsEncryptUpdate(&aes, ciphertext, &outSz, plaintext,
 ↪  sizeof(plaintext));
if (ret != 0) {
    // handle error
}
wc_AesFree(&aes);
```

See: wc_AesCtsDecryptUpdate

Return:

- 0 on successful processing.
- BAD_FUNC_ARG if input arguments are invalid.

C.1.2.44 function wc_AesCtsEncryptFinal

```
int wc_AesCtsEncryptFinal(
    Aes * aes,
    byte * out,
```

```
    word32 * outSz
)
```

This function finalizes the AES CTS encryption operation. It processes any remaining plaintext and completes the encryption.

Parameters:

- **aes** pointer to the Aes structure holding the context of the operation.
- **out** buffer to hold the final encrypted ciphertext. Must be large enough to store any remaining ciphertext from this final step.
- **outSz** size in bytes of the output data written to the out buffer. On input, it should contain the maximum number of bytes that can be written to the out buffer. *Example*

```
Aes aes;
wc_AesInit(&aes, NULL, INVALID_DEVID);
byte key[16] = { 0 };
byte iv[16] = { 0 };
byte plaintext[] = { ... };
byte ciphertext[sizeof(plaintext)];
word32 outSz = sizeof(ciphertext);
wc_AesSetKey(&aes, key, sizeof(key), iv, AES_ENCRYPTION);
// Perform any required update steps using wc_AesCtsEncryptUpdate
int ret = wc_AesCtsEncryptFinal(&aes, ciphertext, &outSz);
if (ret != 0) {
    // handle error
}
wc_AesFree(&aes);
```

See: wc_AesCtsDecryptFinal

Return:

- 0 on successful encryption completion.
- BAD_FUNC_ARG if input arguments are invalid.

C.1.2.45 function wc_AesCtsDecryptUpdate

```
int wc_AesCtsDecryptUpdate(
    Aes * aes,
    byte * out,
    word32 * outSz,
    const byte * in,
    word32 inSz
)
```

This function performs an update step of the AES CTS decryption. It processes a chunk of ciphertext and stores intermediate data.

Parameters:

- **aes** pointer to the Aes structure holding the context of the operation.
- **out** buffer to hold the decrypted plaintext. Must be large enough to store the output from this update step.
- **outSz** size in bytes of the output data written to the out buffer. On input, it should contain the maximum number of bytes that can be written to the out buffer.
- **in** pointer to the ciphertext input data to decrypt.
- **inSz** size of the ciphertext input data in bytes. *Example*

```
Aes aes;
wc_AesInit(&aes, NULL, INVALID_DEVID);
byte key[16] = { 0 };
byte iv[16] = { 0 };
byte ciphertext[] = { ... };
byte plaintext[sizeof(ciphertext)];
word32 outSz = sizeof(plaintext);
wc_AesSetKey(&aes, key, sizeof(key), iv, AES_DECRYPTION);
int ret = wc_AesCtsDecryptUpdate(&aes, plaintext, &outSz, ciphertext,
 ↪  sizeof(ciphertext));
if (ret != 0) {
    // handle error
}
wc_AesFree(&aes);
```

See: wc_AesCtsEncryptUpdate

Return:

- 0 on successful processing.
- BAD_FUNC_ARG if input arguments are invalid.

C.1.2.46 function wc_AesCtsDecryptFinal

```
int wc_AesCtsDecryptFinal(
    Aes * aes,
    byte * out,
    word32 * outSz
)
```

This function finalizes the AES CTS decryption operation. It processes any remaining ciphertext and completes the decryption.

Parameters:

- **aes** pointer to the Aes structure holding the context of the operation.
- **out** buffer to hold the final decrypted plaintext. Must be large enough to store any remaining plaintext from this final step.
- **outSz** size in bytes of the output data written to the out buffer. On input, it should contain the maximum number of bytes that can be written to the out buffer. *Example*

```
Aes aes;
wc_AesInit(&aes, NULL, INVALID_DEVID);
byte key[16] = { 0 };
byte iv[16] = { 0 };
byte ciphertext[] = { ... };
byte plaintext[sizeof(ciphertext)];
word32 outSz = sizeof(plaintext);
wc_AesSetKey(&aes, key, sizeof(key), iv, AES_DECRYPTION);
// Perform any required update steps using wc_AesCtsDecryptUpdate
int ret = wc_AesCtsDecryptFinal(&aes, plaintext, &outSz);
if (ret != 0) {
    // handle error
}
wc_AesFree(&aes);
```

See: wc_AesCtsEncryptFinal

Return:

- 0 on successful decryption completion.
- BAD_FUNC_ARG if input arguments are invalid.

C.1.2.47 function wc_AesCfb1Encrypt

```
int wc_AesCfb1Encrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function encrypts data using AES CFB-1 mode (1-bit feedback). It processes data one bit at a time, making it suitable for bit-oriented applications.

Parameters:

- **aes** pointer to the AES structure containing the key
- **out** pointer to the output buffer to store encrypted data
- **in** pointer to the input buffer containing data to encrypt (packed to left, e.g., 101 is 0x90)
- **sz** size of input in bits

See:

- wc_AesCfb1Decrypt
- wc_AesCfb8Encrypt

Return:

- 0 On success.
- BAD_FUNC_ARG If aes, out, or in is NULL.
- Other negative values on error.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte iv[16] = { }; // initialization vector
byte plaintext[1] = { 0x90 }; // bits 101
byte ciphertext[1];

wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesSetKey(&aes, key, 16, iv, AES_ENCRYPTION);
int ret = wc_AesCfb1Encrypt(&aes, ciphertext, plaintext, 3);
if (ret != 0) {
    // encryption failed
}
wc_AesFree(&aes);
```

C.1.2.48 function wc_AesCfb8Encrypt

```
int wc_AesCfb8Encrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function encrypts data using AES CFB-8 mode (8-bit feedback). It processes data one byte at a time, making it suitable for byte-oriented stream encryption.

Parameters:

- **aes** pointer to the AES structure containing the key
- **out** pointer to the output buffer to store encrypted data
- **in** pointer to the input buffer containing data to encrypt
- **sz** size of input in bytes

See:

- wc_AesCfb8Decrypt
- wc_AesCfb1Encrypt

Return:

- 0 On success.
- BAD_FUNC_ARG If aes, out, or in is NULL.
- Other negative values on error.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte iv[16] = { }; // initialization vector
byte plaintext[10] = { }; // data to encrypt
byte ciphertext[10];

wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesSetKey(&aes, key, 16, iv, AES_ENCRYPTION);
int ret = wc_AesCfb8Encrypt(&aes, ciphertext, plaintext, 10);
if (ret != 0) {
    // encryption failed
}
wc_AesFree(&aes);
```

C.1.2.49 function wc_AesCfb1Decrypt

```
int wc_AesCfb1Decrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function decrypts data using AES CFB-1 mode (1-bit feedback). It processes data one bit at a time, making it suitable for bit-oriented applications.

Parameters:

- **aes** pointer to the AES structure containing the key
- **out** pointer to the output buffer to store decrypted data
- **in** pointer to the input buffer containing data to decrypt
- **sz** size of input in bits

See:

- wc_AesCfb1Encrypt
- wc_AesCfb8Decrypt

Return:

- 0 On success.
- BAD_FUNC_ARG If aes, out, or in is NULL.
- Other negative values on error.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte iv[16] = { }; // initialization vector
byte ciphertext[1] = { }; // encrypted bits
byte plaintext[1];

wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesSetKey(&aes, key, 16, iv, AES_ENCRYPTION);
int ret = wc_AesCfb1Decrypt(&aes, plaintext, ciphertext, 3);
if (ret != 0) {
    // decryption failed
}
wc_AesFree(&aes);
```

C.1.2.50 function wc_AesCfb8Decrypt

```
int wc_AesCfb8Decrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function decrypts data using AES CFB-8 mode (8-bit feedback). It processes data one byte at a time, making it suitable for byte-oriented stream decryption.

Parameters:

- **aes** pointer to the AES structure containing the key
- **out** pointer to the output buffer to store decrypted data
- **in** pointer to the input buffer containing data to decrypt
- **sz** size of input in bytes

See:

- wc_AesCfb8Encrypt
- wc_AesCfb1Decrypt

Return:

- 0 On success.
- BAD_FUNC_ARG If aes, out, or in is NULL.
- Other negative values on error.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte iv[16] = { }; // initialization vector
byte ciphertext[10] = { }; // encrypted data
byte plaintext[10];
```

```
wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesSetKey(&aes, key, 16, iv, AES_ENCRYPTION);
int ret = wc_AesCfb8Decrypt(&aes, plaintext, ciphertext, 10);
if (ret != 0) {
    // decryption failed
}
wc_AesFree(&aes);
```

C.1.2.51 function wc_AesOfbEncrypt

```
int wc_AesOfbEncrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function encrypts data using AES OFB mode (Output Feedback). OFB mode turns a block cipher into a stream cipher by encrypting the IV and XORing with plaintext.

Parameters:

- **aes** pointer to the AES structure containing the key
- **out** pointer to the output buffer to store encrypted data
- **in** pointer to the input buffer containing data to encrypt
- **sz** size of input in bytes

See:

- wc_AesOfbDecrypt
- wc_AesSetKey

Return:

- 0 On success.
- BAD_FUNC_ARG If aes, out, or in is NULL.
- Other negative values on error.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte iv[16] = { }; // initialization vector
byte plaintext[100] = { }; // data to encrypt
byte ciphertext[100];

wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesSetKey(&aes, key, 16, iv, AES_ENCRYPTION);
int ret = wc_AesOfbEncrypt(&aes, ciphertext, plaintext, 100);
if (ret != 0) {
    // encryption failed
}
wc_AesFree(&aes);
```

C.1.2.52 function wc_AesOfbDecrypt

```
int wc_AesOfbDecrypt(
    Aes * aes,
    byte * out,
```

```
    const byte * in,
    word32 sz
)
```

This function decrypts data using AES OFB mode (Output Feedback). In OFB mode, encryption and decryption are the same operation.

Parameters:

- **aes** pointer to the AES structure containing the key
- **out** pointer to the output buffer to store decrypted data
- **in** pointer to the input buffer containing data to decrypt
- **sz** size of input in bytes

See:

- wc_AesOfbEncrypt
- wc_AesSetKey

Return:

- 0 On success.
- BAD_FUNC_ARG If aes, out, or in is NULL.
- Other negative values on error.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte iv[16] = { }; // initialization vector
byte ciphertext[100] = { }; // encrypted data
byte plaintext[100];

wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesSetKey(&aes, key, 16, iv, AES_ENCRYPTION);
int ret = wc_AesOfbDecrypt(&aes, plaintext, ciphertext, 100);
if (ret != 0) {
    // decryption failed
}
wc_AesFree(&aes);
```

C.1.2.53 function wc_AesEcbEncrypt

```
int wc_AesEcbEncrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function encrypts data using AES ECB mode (Electronic Codebook). Warning: ECB mode is not recommended for most use cases as it does not provide semantic security. Each block is encrypted independently.

Parameters:

- **aes** pointer to the AES structure containing the key
- **out** pointer to the output buffer to store encrypted data
- **in** pointer to the input buffer containing data to encrypt
- **sz** size of input in bytes (must be multiple of AES_BLOCK_SIZE)

See:

- wc_AesEcbDecrypt
- wc_AesSetKey

Return:

- 0 On success.
- BAD_FUNC_ARG If aes, out, or in is NULL.
- Other negative values on error.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte plaintext[32] = { }; // data to encrypt
byte ciphertext[32];

wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesSetKey(&aes, key, 16, NULL, AES_ENCRYPTION);
int ret = wc_AesEcbEncrypt(&aes, ciphertext, plaintext, 32);
if (ret != 0) {
    // encryption failed
}
wc_AesFree(&aes);
```

C.1.2.54 function wc_AesEcbDecrypt

```
int wc_AesEcbDecrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function decrypts data using AES ECB mode (Electronic Codebook). Warning: ECB mode is not recommended for most use cases as it does not provide semantic security. Each block is decrypted independently.

Parameters:

- **aes** pointer to the AES structure containing the key
- **out** pointer to the output buffer to store decrypted data
- **in** pointer to the input buffer containing data to decrypt
- **sz** size of input in bytes (must be multiple of AES_BLOCK_SIZE)

See:

- wc_AesEcbEncrypt
- wc_AesSetKey

Return:

- 0 On success.
- BAD_FUNC_ARG If aes, out, or in is NULL.
- Other negative values on error.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
```

```
byte ciphertext[32] = { }; // encrypted data
byte plaintext[32];

wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesSetKey(&aes, key, 16, NULL, AES_DECRYPTION);
int ret = wc_AesEcbDecrypt(&aes, plaintext, ciphertext, 32);
if (ret != 0) {
    // decryption failed
}
wc_AesFree(&aes);
```

C.1.2.55 function wc_AesCtrSetKey

```
int wc_AesCtrSetKey(
    Aes * aes,
    const byte * key,
    word32 len,
    const byte * iv,
    int dir
)
```

This function sets the key and IV for AES CTR mode. It initializes the AES structure for counter mode encryption or decryption.

Parameters:

- **aes** pointer to the AES structure to initialize
- **key** pointer to the key buffer (16, 24, or 32 bytes)
- **len** length of the key in bytes
- **iv** pointer to the initialization vector (16 bytes)
- **dir** cipher direction (always use AES_ENCRYPTION for CTR mode)

See:

- wc_AesCtrEncrypt
- wc_AesSetKey

Return:

- 0 On success.
- BAD_FUNC_ARG If aes, key, or iv is NULL, or if key length is invalid.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte iv[16] = { }; // initialization vector

wc_AesInit(&aes, NULL, INVALID_DEVID);
int ret = wc_AesCtrSetKey(&aes, key, 16, iv, AES_ENCRYPTION);
if (ret != 0) {
    // failed to set key
}
wc_AesFree(&aes);
```

C.1.2.56 function wc_AesGcmSetKey_ex

```
int wc_AesGcmSetKey_ex(
    Aes * aes,
```

```
    const byte * key,
    word32 len,
    word32 kup
)
```

This function sets the key for AES GCM with an extended key update parameter. It allows for key updates in certain hardware implementations.

Parameters:

- **aes** pointer to the AES structure to initialize
- **key** pointer to the key buffer (16, 24, or 32 bytes)
- **len** length of the key in bytes
- **kup** key update parameter for hardware implementations

See:

- wc_AesGcmSetKey
- wc_AesGcmInit

Return:

- 0 On success.
- BAD_FUNC_ARG If aes or key is NULL, or if key length is invalid.

Note: This function is currently only available when building with Xilinx hardware acceleration. It requires one of the following build options: WOLFSSL_XILINX_CRYPT (for Xilinx SecureIP integration) or WOLFSSL_AFALG_XILINX_AES (for Xilinx AF_ALG support). This API may be exposed for additional build configurations in the future.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key

wc_AesInit(&aes, NULL, INVALID_DEVID);
int ret = wc_AesGcmSetKey_ex(&aes, key, 16, 0);
if (ret != 0) {
    // failed to set key
}
wc_AesFree(&aes);
```

C.1.2.57 function wc_AesGcmInit

```
int wc_AesGcmInit(
    Aes * aes,
    const byte * key,
    word32 len,
    const byte * iv,
    word32 ivSz
)
```

This function initializes an AES GCM cipher with key and IV. It can be called with NULL key to only set the IV, or with NULL IV to only set the key.

Parameters:

- **aes** pointer to the AES structure to initialize
- **key** pointer to the key buffer, or NULL to skip key setting
- **len** length of the key in bytes
- **iv** pointer to the IV/nonce buffer, or NULL to skip IV setting

- **ivSz** length of the IV/nonce in bytes

See:

- wc_AesGcmSetKey
- wc_AesGcmEncrypt

Return:

- 0 On success.
- BAD_FUNC_ARG If aes is NULL, or if parameters are invalid.
- MEMORY_E If dynamic memory allocation fails.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte iv[12] = { }; // 96-bit nonce

wc_AesInit(&aes, NULL, INVALID_DEVID);
int ret = wc_AesGcmInit(&aes, key, 16, iv, 12);
if (ret != 0) {
    // failed to initialize
}
wc_AesFree(&aes);
```

C.1.2.58 function wc_AesGcmEncryptInit

```
int wc_AesGcmEncryptInit(
    Aes * aes,
    const byte * key,
    word32 len,
    const byte * iv,
    word32 ivSz
)
```

This function initializes an AES GCM cipher for encryption. It is a convenience wrapper around wc_AesGcmInit for encryption operations.

Parameters:

- **aes** pointer to the AES structure to initialize
- **key** pointer to the key buffer, or NULL to skip key setting
- **len** length of the key in bytes
- **iv** pointer to the IV/nonce buffer, or NULL to skip IV setting
- **ivSz** length of the IV/nonce in bytes

See:

- wc_AesGcmInit
- wc_AesGcmEncryptUpdate

Return:

- 0 On success.
- BAD_FUNC_ARG If aes is NULL, or if parameters are invalid.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte iv[12] = { }; // 96-bit nonce
```

```
wc_AesInit(&aes, NULL, INVALID_DEVID);
int ret = wc_AesGcmEncryptInit(&aes, key, 16, iv, 12);
if (ret != 0) {
    // failed to initialize
}
wc_AesFree(&aes);
```

C.1.2.59 function wc_AesGcmEncryptInit_ex

```
int wc_AesGcmEncryptInit_ex(
    Aes * aes,
    const byte * key,
    word32 len,
    byte * ivOut,
    word32 ivOutSz
)
```

This function initializes an AES GCM cipher for encryption and outputs the IV. This is useful when part of the IV is generated internally. Must call wc_AesGcmSetIV() before this function to set the fixed part of the IV.

Parameters:

- **aes** pointer to the AES structure to initialize
- **key** pointer to the key buffer, or NULL to skip key setting
- **len** length of the key in bytes
- **ivOut** pointer to buffer to receive the complete IV
- **ivOutSz** length of the IV output buffer in bytes

See:

- wc_AesGcmSetIV
- wc_AesGcmEncryptUpdate

Return:

- 0 On success.
- BAD_FUNC_ARG If aes, ivOut is NULL, or if ivOutSz doesn't match the cached nonce size.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte ivFixed[4] = { }; // fixed part of IV
byte ivOut[12];
WC_RNG rng;

wc_InitRng(&rng);
wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesGcmSetIV(&aes, 12, ivFixed, 4, &rng);
int ret = wc_AesGcmEncryptInit_ex(&aes, key, 16, ivOut, 12);
if (ret != 0) {
    // failed to initialize
}
wc_AesFree(&aes);
wc_FreeRng(&rng);
```

C.1.2.60 function wc_AesGcmEncryptUpdate

```
int wc_AesGcmEncryptUpdate(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    const byte * authIn,
    word32 authInSz
)
```

This function performs an update step of AES GCM encryption. It processes plaintext and/or additional authentication data (AAD) in a streaming fashion.

Parameters:

- **aes** pointer to the AES structure
- **out** pointer to buffer to store ciphertext (can be NULL if sz=0)
- **in** pointer to plaintext to encrypt (can be NULL if sz=0)
- **sz** length of plaintext in bytes
- **authIn** pointer to additional authentication data (can be NULL)
- **authInSz** length of AAD in bytes

See:

- wc_AesGcmInit
- wc_AesGcmEncryptInit
- wc_AesGcmEncryptFinal

Return:

- 0 On success.
- BAD_FUNC_ARG If aes is NULL, or a length is non-zero but buffer is NULL.

All the AAD must be passed to update before the plaintext. The last part of AAD can be passed with the first part of plaintext.

Must set key and IV before calling this function. Must call wc_AesGcmInit() before calling this function.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte iv[12] = { }; // nonce
byte plaintext[100] = { }; // data
byte ciphertext[100];
byte aad[20] = { }; // additional data

wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesGcmInit(&aes, key, 16, iv, 12);
int ret = wc_AesGcmEncryptUpdate(&aes, ciphertext, plaintext, 100,
                                 aad, 20);
if (ret != 0) {
    // encryption failed
}
wc_AesFree(&aes);
```

C.1.2.61 function wc_AesGcmEncryptFinal

```
int wc_AesGcmEncryptFinal(
    Aes * aes,
    byte * authTag,
    word32 authTagSz
)
```

This function finalizes AES GCM encryption and generates the authentication tag. This must be called after all data has been processed with wc_AesGcmEncryptUpdate.

Parameters:

- **aes** pointer to the AES structure
- **authTag** pointer to buffer to store the authentication tag
- **authTagSz** length of the authentication tag in bytes (typically 12 or 16)

See:

- wc_AesGcmEncryptUpdate
- wc_AesGcmDecryptFinal

Return:

- 0 On success.
- BAD_FUNC_ARG If aes or authTag is NULL, or if authTagSz is invalid.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte iv[12] = { }; // nonce
byte plaintext[100] = { }; // data
byte ciphertext[100];
byte authTag[16];

wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesGcmEncryptInit(&aes, key, 16, iv, 12);
wc_AesGcmEncryptUpdate(&aes, ciphertext, plaintext, 100, NULL, 0);
int ret = wc_AesGcmEncryptFinal(&aes, authTag, 16);
if (ret != 0) {
    // failed to generate tag
}
wc_AesFree(&aes);
```

C.1.2.62 function wc_AesGcmDecryptInit

```
int wc_AesGcmDecryptInit(
    Aes * aes,
    const byte * key,
    word32 len,
    const byte * iv,
    word32 ivSz
)
```

This function initializes an AES GCM cipher for decryption. It is a convenience wrapper around wc_AesGcmInit for decryption operations.

Parameters:

- **aes** pointer to the AES structure to initialize
- **key** pointer to the key buffer, or NULL to skip key setting

- **len** length of the key in bytes
- **iv** pointer to the IV/nonce buffer, or NULL to skip IV setting
- **ivSz** length of the IV/nonce in bytes

See:

- wc_AesGcmInit
- wc_AesGcmDecryptUpdate

Return:

- 0 On success.
- BAD_FUNC_ARG If aes is NULL, or if parameters are invalid.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte iv[12] = { }; // 96-bit nonce

wc_AesInit(&aes, NULL, INVALID_DEVID);
int ret = wc_AesGcmDecryptInit(&aes, key, 16, iv, 12);
if (ret != 0) {
    // failed to initialize
}
wc_AesFree(&aes);
```

C.1.2.63 function wc_AesGcmDecryptUpdate

```
int wc_AesGcmDecryptUpdate(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    const byte * authIn,
    word32 authInSz
)
```

This function performs an update step of AES GCM decryption. It processes ciphertext and/or additional authentication data (AAD) in a streaming fashion.

Parameters:

- **aes** pointer to the AES structure
- **out** pointer to buffer to store plaintext (can be NULL if sz=0)
- **in** pointer to ciphertext to decrypt (can be NULL if sz=0)
- **sz** length of ciphertext in bytes
- **authIn** pointer to additional authentication data (can be NULL)
- **authInSz** length of AAD in bytes

See:

- wc_AesGcmInit
- wc_AesGcmDecryptInit
- wc_AesGcmDecryptFinal

Return:

- 0 On success.
- BAD_FUNC_ARG If aes is NULL, or a length is non-zero but buffer is NULL.

All the AAD must be passed to update before the ciphertext. The last part of AAD can be passed with the first part of ciphertext.

Must set key and IV before calling this function. Must call wc_AesGcmInit() before calling this function.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte iv[12] = { }; // nonce
byte ciphertext[100] = { }; // encrypted data
byte plaintext[100];
byte aad[20] = { }; // additional data

wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesGcmInit(&aes, key, 16, iv, 12);
int ret = wc_AesGcmDecryptUpdate(&aes, plaintext, ciphertext, 100,
                                 aad, 20);
if (ret != 0) {
    // decryption failed
}
wc_AesFree(&aes);
```

C.1.2.64 function wc_AesGcmDecryptFinal

```
int wc_AesGcmDecryptFinal(
    Aes * aes,
    const byte * authTag,
    word32 authTagSz
)
```

This function finalizes AES GCM decryption and verifies the authentication tag. This must be called after all data has been processed with wc_AesGcmDecryptUpdate.

Parameters:

- **aes** pointer to the AES structure
- **authTag** pointer to the authentication tag to verify
- **authTagSz** length of the authentication tag in bytes

See:

- wc_AesGcmDecryptUpdate
- wc_AesGcmEncryptFinal

Return:

- 0 On success.
- AES_GCM_AUTH_E If authentication tag verification fails.
- BAD_FUNC_ARG If aes or authTag is NULL, or if authTagSz is invalid.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte iv[12] = { }; // nonce
byte ciphertext[100] = { }; // encrypted data
byte plaintext[100];
byte authTag[16] = { }; // received tag
```

```
wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesGcmDecryptInit(&aes, key, 16, iv, 12);
wc_AesGcmDecryptUpdate(&aes, plaintext, ciphertext, 100, NULL, 0);
int ret = wc_AesGcmDecryptFinal(&aes, authTag, 16);
if (ret != 0) {
    // authentication failed
}
wc_AesFree(&aes);
```

C.1.2.65 function wc_AesGcmSetExtIV

```
int wc_AesGcmSetExtIV(
    Aes * aes,
    const byte * iv,
    word32 ivSz
)
```

This function sets an external IV for AES GCM. This allows using an IV that was generated externally or received from another source.

Parameters:

- **aes** pointer to the AES structure
- **iv** pointer to the IV/nonce buffer
- **ivSz** length of the IV/nonce in bytes

See:

- wc_AesGcmSetIV
- wc_AesGcmInit

Return:

- 0 On success.
- BAD_FUNC_ARG If aes or iv is NULL, or if ivSz is invalid.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte iv[12] = { }; // external nonce

wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesGcmSetKey(&aes, key, 16);
int ret = wc_AesGcmSetExtIV(&aes, iv, 12);
if (ret != 0) {
    // failed to set IV
}
wc_AesFree(&aes);
```

C.1.2.66 function wc_AesGcmSetIV

```
int wc_AesGcmSetIV(
    Aes * aes,
    word32 ivSz,
    const byte * ivFixed,
    word32 ivFixedSz,
    WC_RNG * rng
)
```

This function sets the IV for AES GCM with optional random generation. It can generate part of the IV using an RNG, which is useful for ensuring IV uniqueness.

Parameters:

- **aes** pointer to the AES structure
- **ivSz** total length of the IV/nonce in bytes
- **ivFixed** pointer to the fixed part of the IV (can be NULL)
- **ivFixedSz** length of the fixed part in bytes
- **rng** pointer to initialized RNG for generating random part (can be NULL if ivFixedSz equals ivSz)

See:

- wc_AesGcmSetExtIV
- wc_AesGcmEncryptInit_ex

Return:

- 0 On success.
- BAD_FUNC_ARG If aes is NULL, or if parameters are invalid.
- Other negative values on RNG or other errors.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte ivFixed[4] = { }; // fixed part
WC_RNG rng;

wc_InitRng(&rng);
wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesGcmSetKey(&aes, key, 16);
int ret = wc_AesGcmSetIV(&aes, 12, ivFixed, 4, &rng);
if (ret != 0) {
    // failed to set IV
}
wc_AesFree(&aes);
wc_FreeRng(&rng);
```

C.1.2.67 function wc_AesGcmEncrypt_ex

```
int wc_AesGcmEncrypt_ex(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    byte * ivOut,
    word32 ivOutSz,
    byte * authTag,
    word32 authTagSz,
    const byte * authIn,
    word32 authInSz
)
```

This function performs AES GCM encryption with extended parameters, including IV output. This is a one-shot encryption function that outputs the generated IV.

Parameters:

- **aes** pointer to the AES structure

- **out** pointer to buffer to store ciphertext
- **in** pointer to plaintext to encrypt
- **sz** length of plaintext in bytes
- **ivOut** pointer to buffer to receive the IV
- **ivOutSz** length of the IV output buffer in bytes
- **authTag** pointer to buffer to store authentication tag
- **authTagSz** length of authentication tag in bytes
- **authIn** pointer to additional authentication data
- **authInSz** length of AAD in bytes

See:

- wc_AesGcmEncrypt
- wc_AesGcmSetIV

Return:

- 0 On success.
- BAD_FUNC_ARG If parameters are invalid.
- WC_PENDING_E If submitted to an asynchronous device and still in progress.
- Other negative values on error.

The IV is taken from an internal counter that is advanced whenever an IV is consumed - on success, and on submission to an asynchronous device - so no two encryptions under one key use the same IV. A call that fails before the cipher consumes the IV leaves the counter where it was. ivOut must not overlap out, in, authTag or authIn: the buffer is the working copy of the IV that was consumed, and overwriting it corrupts the counter for the next call. ivOut may be written even when the function returns an error.

When the Aes carries an asynchronous device, a return of WC_PENDING_E means the operation was submitted, not that it finished. ivOut - like out, in, authTag and authIn - must stay allocated and unmodified until the operation completes, because the backend reads the IV from it at completion time.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte ivFixed[4] = { }; // fixed part
byte ivOut[12];
byte plaintext[100] = { }; // data
byte ciphertext[100];
byte authTag[16];
WC_RNG rng;

wc_InitRng(&rng);
wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesGcmSetKey(&aes, key, 16);
wc_AesGcmSetIV(&aes, 12, ivFixed, 4, &rng);
int ret = wc_AesGcmEncrypt_ex(&aes, ciphertext, plaintext, 100,
                              ivOut, 12, authTag, 16, NULL, 0);
if (ret != 0) {
    // encryption failed
}
wc_AesFree(&aes);
wc_FreeRng(&rng);
```

C.1.2.68 function wc_Gmac

```
int wc_Gmac(
    const byte * key,
    word32 keySz,
    byte * iv,
    word32 ivSz,
    const byte * authIn,
    word32 authInSz,
    byte * authTag,
    word32 authTagSz,
    WC_RNG * rng
)
```

This function performs GMAC (Galois Message Authentication Code) generation. GMAC is essentially AES-GCM with no plaintext, used for authentication only.

Parameters:

- **key** pointer to the key buffer
- **keySz** length of the key in bytes (16, 24, or 32)
- **iv** pointer to the IV/nonce buffer
- **ivSz** length of the IV/nonce in bytes
- **authIn** pointer to data to authenticate
- **authInSz** length of data to authenticate in bytes
- **authTag** pointer to buffer to store authentication tag
- **authTagSz** length of authentication tag in bytes
- **rng** pointer to initialized RNG (can be NULL if IV is complete)

See:

- wc_GmacVerify
- wc_AesGcmEncrypt

Return:

- 0 On success.
- BAD_FUNC_ARG If parameters are invalid.
- Other negative values on error.

Example

```
byte key[16] = { }; // 128-bit key
byte iv[12] = { }; // nonce
byte data[100] = { }; // data to authenticate
byte authTag[16];

int ret = wc_Gmac(key, 16, iv, 12, data, 100, authTag, 16, NULL);
if (ret != 0) {
    // GMAC generation failed
}
```

C.1.2.69 function wc_GmacVerify

```
int wc_GmacVerify(
    const byte * key,
    word32 keySz,
    const byte * iv,
    word32 ivSz,
    const byte * authIn,
    word32 authInSz,
```

```
    const byte * authTag,
    word32 authTagSz
)
```

This function verifies a GMAC (Galois Message Authentication Code). It computes the GMAC and compares it with the provided tag.

Parameters:

- **key** pointer to the key buffer
- **keySz** length of the key in bytes (16, 24, or 32)
- **iv** pointer to the IV/nonce buffer
- **ivSz** length of the IV/nonce in bytes
- **authIn** pointer to data to authenticate
- **authInSz** length of data to authenticate in bytes
- **authTag** pointer to the authentication tag to verify
- **authTagSz** length of authentication tag in bytes

See:

- wc_Gmac
- wc_AesGcmDecrypt

Return:

- 0 On successful verification.
- AES_GCM_AUTH_E If authentication tag verification fails.
- BAD_FUNC_ARG If parameters are invalid.
- Other negative values on error.

Example

```
byte key[16] = { }; // 128-bit key
byte iv[12] = { }; // nonce
byte data[100] = { }; // data to authenticate
byte authTag[16] = { }; // received tag

int ret = wc_GmacVerify(key, 16, iv, 12, data, 100, authTag, 16);
if (ret != 0) {
    // GMAC verification failed
}
```

C.1.2.70 function wc_AesCcmSetNonce

```
int wc_AesCcmSetNonce(
    Aes * aes,
    const byte * nonce,
    word32 nonceSz
)
```

This function sets the nonce for AES CCM mode. The nonce must be set before encryption or decryption operations.

Parameters:

- **aes** pointer to the AES structure
- **nonce** pointer to the nonce buffer
- **nonceSz** length of the nonce in bytes (7-13 bytes for CCM)

See:

- wc_AesCcmEncrypt
- wc_AesCcmSetKey

Return:

- 0 On success.
- BAD_FUNC_ARG If aes or nonce is NULL, or if nonceSz is invalid.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte nonce[12] = { }; // nonce

wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesCcmSetKey(&aes, key, 16);
int ret = wc_AesCcmSetNonce(&aes, nonce, 12);
if (ret != 0) {
    // failed to set nonce
}
wc_AesFree(&aes);
```

C.1.2.71 function wc_AesCcmEncrypt_ex

```
int wc_AesCcmEncrypt_ex(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    byte * ivOut,
    word32 ivOutSz,
    byte * authTag,
    word32 authTagSz,
    const byte * authIn,
    word32 authInSz
)
```

This function performs AES CCM encryption with extended parameters, including nonce output. This is useful when part of the nonce is generated internally.

Parameters:

- **aes** pointer to the AES structure
- **out** pointer to buffer to store ciphertext
- **in** pointer to plaintext to encrypt
- **sz** length of plaintext in bytes
- **ivOut** pointer to buffer to receive the nonce
- **ivOutSz** length of the nonce output buffer in bytes
- **authTag** pointer to buffer to store authentication tag
- **authTagSz** length of authentication tag in bytes
- **authIn** pointer to additional authentication data
- **authInSz** length of AAD in bytes

See:

- wc_AesCcmEncrypt
- wc_AesCcmSetNonce

Return:

- 0 On success.
- BAD_FUNC_ARG If parameters are invalid.
- Other negative values on error.

The nonce is taken from an internal counter that is advanced whenever a nonce is consumed - on success, and on submission to an asynchronous device - so no two encryptions under one key use the same nonce. A call that fails before the cipher consumes the nonce leaves the counter where it was. ivOut must not overlap out, in, authTag or authIn: the buffer is the working copy of the nonce that was consumed, and overwriting it corrupts the counter for the next call. ivOut may be written even when the function returns an error - earlier releases left it untouched on failure.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
byte nonce[12];
byte plaintext[100] = { }; // data
byte ciphertext[100];
byte authTag[16];

wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesCcmSetKey(&aes, key, 16);
int ret = wc_AesCcmEncrypt_ex(&aes, ciphertext, plaintext, 100,
                              nonce, 12, authTag, 16, NULL, 0);
if (ret != 0) {
    // encryption failed
}
wc_AesFree(&aes);
```

C.1.2.72 function wc_AesKeyWrap

```
int wc_AesKeyWrap(
    const byte * key,
    word32 keySz,
    const byte * in,
    word32 inSz,
    byte * out,
    word32 outSz,
    const byte * iv
)
```

This function wraps a key using AES Key Wrap algorithm (RFC 3394). This is commonly used to securely transport cryptographic keys.

Parameters:

- **key** pointer to the key-encryption key
- **keySz** length of the key-encryption key in bytes
- **in** pointer to the key to wrap
- **inSz** length of the key to wrap in bytes
- **out** pointer to buffer to store wrapped key
- **outSz** size of output buffer in bytes
- **iv** pointer to IV (typically NULL to use default)

See:

- wc_AesKeyUnWrap
- wc_AesKeyWrap_ex

Return:

- Length of wrapped key in bytes on success.
- BAD_FUNC_ARG If parameters are invalid.
- Other negative values on error.

Example

```
byte kek[16] = { }; // key-encryption key
byte keyToWrap[16] = { }; // key to wrap
byte wrappedKey[24];

int wrappedLen = wc_AesKeyWrap(kek, 16, keyToWrap, 16, wrappedKey,
                               24, NULL);
if (wrappedLen <= 0) {
    // key wrap failed
}
```

C.1.2.73 function wc_AesKeyWrap_ex

```
int wc_AesKeyWrap_ex(
    Aes * aes,
    const byte * in,
    word32 inSz,
    byte * out,
    word32 outSz,
    const byte * iv
)
```

This function wraps a key using AES Key Wrap algorithm with an initialized AES structure. This allows reusing the same AES structure for multiple wrap operations.

Parameters:

- **aes** pointer to initialized AES structure
- **in** pointer to the key to wrap
- **inSz** length of the key to wrap in bytes
- **out** pointer to buffer to store wrapped key
- **outSz** size of output buffer in bytes
- **iv** pointer to IV (typically NULL to use default)

See:

- wc_AesKeyWrap
- wc_AesKeyUnWrap_ex

Return:

- Length of wrapped key in bytes on success.
- BAD_FUNC_ARG If parameters are invalid.
- Other negative values on error.

Example

```
Aes aes;
byte kek[16] = { }; // key-encryption key
byte keyToWrap[16] = { }; // key to wrap
byte wrappedKey[24];

wc_AesInit(&aes, NULL, INVALID_DEVID);
```

```
wc_AesSetKey(&aes, kek, 16, NULL, AES_ENCRYPTION);
int wrappedLen = wc_AesKeyWrap_ex(&aes, keyToWrap, 16, wrappedKey,
                                  24, NULL);
if (wrappedLen <= 0) {
    // key wrap failed
}
wc_AesFree(&aes);
```

C.1.2.74 function wc_AesKeyUnWrap

```
int wc_AesKeyUnWrap(
    const byte * key,
    word32 keySz,
    const byte * in,
    word32 inSz,
    byte * out,
    word32 outSz,
    const byte * iv
)
```

This function unwraps a key using AES Key Unwrap algorithm (RFC 3394). This is used to securely receive cryptographic keys that were wrapped.

Parameters:

- **key** pointer to the key-encryption key
- **keySz** length of the key-encryption key in bytes
- **in** pointer to the wrapped key
- **inSz** length of the wrapped key in bytes
- **out** pointer to buffer to store unwrapped key
- **outSz** size of output buffer in bytes
- **iv** pointer to IV (typically NULL to use default)

See:

- wc_AesKeyWrap
- wc_AesKeyUnWrap_ex

Return:

- Length of unwrapped key in bytes on success.
- BAD_FUNC_ARG If parameters are invalid.
- Other negative values on error.

Example

```
byte kek[16] = { }; // key-encryption key
byte wrappedKey[24] = { }; // wrapped key
byte unwrappedKey[16];

int unwrappedLen = wc_AesKeyUnWrap(kek, 16, wrappedKey, 24,
                                   unwrappedKey, 16, NULL);
if (unwrappedLen <= 0) {
    // key unwrap failed
}
```

C.1.2.75 function wc_AesKeyUnWrap_ex

```
int wc_AesKeyUnWrap_ex(
    Aes * aes,
    const byte * in,
    word32 inSz,
    byte * out,
    word32 outSz,
    const byte * iv
)
```

This function unwraps a key using AES Key Unwrap algorithm with an initialized AES structure. This allows reusing the same AES structure for multiple unwrap operations.

Parameters:

- **aes** pointer to initialized AES structure
- **in** pointer to the wrapped key
- **inSz** length of the wrapped key in bytes
- **out** pointer to buffer to store unwrapped key
- **outSz** size of output buffer in bytes
- **iv** pointer to IV (typically NULL to use default)

See:

- wc_AesKeyUnWrap
- wc_AesKeyWrap_ex

Return:

- Length of unwrapped key in bytes on success.
- BAD_FUNC_ARG If parameters are invalid.
- Other negative values on error.

Example

```
Aes aes;
byte kek[16] = { }; // key-encryption key
byte wrappedKey[24] = { }; // wrapped key
byte unwrappedKey[16];

wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesSetKey(&aes, kek, 16, NULL, AES_ENCRYPTION);
int unwrappedLen = wc_AesKeyUnWrap_ex(&aes, wrappedKey, 24,
                                      unwrappedKey, 16, NULL);
if (unwrappedLen <= 0) {
    // key unwrap failed
}
wc_AesFree(&aes);
```

C.1.2.76 function wc_AesXtsEncryptConsecutiveSectors

```
int wc_AesXtsEncryptConsecutiveSectors(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    word64 sector,
    word32 sectorSz
)
```

This function encrypts multiple consecutive sectors using AES XTS mode. It processes multiple sectors in sequence, automatically incrementing the sector number for each sector.

Parameters:

- **aes** pointer to the XtsAes structure
- **out** pointer to buffer to store encrypted data
- **in** pointer to plaintext data to encrypt
- **sz** total length of data in bytes
- **sector** starting sector number for the tweak
- **sectorSz** size of each sector in bytes

See:

- wc_AesXtsDecryptConsecutiveSectors
- wc_AesXtsEncryptSector

Return:

- 0 On success.
- BAD_FUNC_ARG If aes, out, or in is NULL, or if sectorSz is 0, or if sz is less than AES_BLOCK_SIZE.
- Other negative values on error.

Example

```
XtsAes aes;
byte key[32] = { }; // 256-bit key
byte plaintext[1024] = { }; // data
byte ciphertext[1024];

wc_AesXtsSetKey(&aes, key, 32, AES_ENCRYPTION, NULL, INVALID_DEVID);
int ret = wc_AesXtsEncryptConsecutiveSectors(&aes, ciphertext,
                                             plaintext, 1024, 0, 512);
if (ret != 0) {
    // encryption failed
}
wc_AesXtsFree(&aes);
```

C.1.2.77 function wc_AesXtsDecryptConsecutiveSectors

```
int wc_AesXtsDecryptConsecutiveSectors(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    word64 sector,
    word32 sectorSz
)
```

This function decrypts multiple consecutive sectors using AES XTS mode. It processes multiple sectors in sequence, automatically incrementing the sector number for each sector.

Parameters:

- **aes** pointer to the XtsAes structure
- **out** pointer to buffer to store decrypted data
- **in** pointer to ciphertext data to decrypt
- **sz** total length of data in bytes
- **sector** starting sector number for the tweak
- **sectorSz** size of each sector in bytes

See:

- wc_AesXtsEncryptConsecutiveSectors
- wc_AesXtsDecryptSector

Return:

- 0 On success.
- BAD_FUNC_ARG If aes, out, or in is NULL, or if sectorSz is 0, or if sz is less than AES_BLOCK_SIZE.
- Other negative values on error.

Example

```
XtsAes aes;
byte key[32] = { }; // 256-bit key
byte ciphertext[1024] = { }; // encrypted data
byte plaintext[1024];

wc_AesXtsSetKey(&aes, key, 32, AES_DECRYPTION, NULL, INVALID_DEVID);
int ret = wc_AesXtsDecryptConsecutiveSectors(&aes, plaintext,
                                             ciphertext, 1024, 0, 512);
if (ret != 0) {
    // decryption failed
}
wc_AesXtsFree(&aes);
```

C.1.2.78 function wc_AesXtsEncryptInit

```
int wc_AesXtsEncryptInit(
    XtsAes * aes,
    const byte * i,
    word32 iSz,
    struct XtsAesStreamData * stream
)
```

This function initializes streaming AES XTS encryption. It sets up the context for processing data in multiple update calls.

Parameters:

- **aes** pointer to the XtsAes structure
- **i** pointer to the tweak/IV buffer
- **iSz** length of the tweak/IV in bytes
- **stream** pointer to XtsAesStreamData structure for streaming state

See:

- wc_AesXtsEncryptUpdate
- wc_AesXtsEncryptFinal

Return:

- 0 On success.
- BAD_FUNC_ARG If parameters are invalid.

Example

```
XtsAes aes;
struct XtsAesStreamData stream;
byte key[32] = { }; // 256-bit key
byte tweak[16] = { }; // tweak value
```

```
wc_AesXtsSetKey(&aes, key, 32, AES_ENCRYPTION, NULL, INVALID_DEVID);
int ret = wc_AesXtsEncryptInit(&aes, tweak, 16, &stream);
if (ret != 0) {
    // initialization failed
}
wc_AesXtsFree(&aes);
```

C.1.2.79 function wc_AesXtsDecryptInit

```
int wc_AesXtsDecryptInit(
    XtsAes * aes,
    const byte * i,
    word32 iSz,
    struct XtsAesStreamData * stream
)
```

This function initializes streaming AES XTS decryption. It sets up the context for processing data in multiple update calls.

Parameters:

- **aes** pointer to the XtsAes structure
- **i** pointer to the tweak/IV buffer
- **iSz** length of the tweak/IV in bytes
- **stream** pointer to XtsAesStreamData structure for streaming state

See:

- wc_AesXtsDecryptUpdate
- wc_AesXtsDecryptFinal

Return:

- 0 On success.
- BAD_FUNC_ARG If parameters are invalid.

Example

```
XtsAes aes;
struct XtsAesStreamData stream;
byte key[32] = { }; // 256-bit key
byte tweak[16] = { }; // tweak value

wc_AesXtsSetKey(&aes, key, 32, AES_DECRYPTION, NULL, INVALID_DEVID);
int ret = wc_AesXtsDecryptInit(&aes, tweak, 16, &stream);
if (ret != 0) {
    // initialization failed
}
wc_AesXtsFree(&aes);
```

C.1.2.80 function wc_AesXtsEncryptUpdate

```
int wc_AesXtsEncryptUpdate(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
```

```
    struct XtsAesStreamData * stream
)
```

This function performs an update step of streaming AES XTS encryption. It processes a chunk of data and can be called multiple times.

Parameters:

- **aes** pointer to the XtsAes structure
- **out** pointer to buffer to store encrypted data
- **in** pointer to plaintext data to encrypt
- **sz** length of data in bytes
- **stream** pointer to XtsAesStreamData structure for streaming state

See:

- wc_AesXtsEncryptInit
- wc_AesXtsEncryptFinal

Return:

- 0 On success.
- BAD_FUNC_ARG If parameters are invalid.

Example

```
XtsAes aes;
struct XtsAesStreamData stream;
byte key[32] = { }; // 256-bit key
byte tweak[16] = { }; // tweak value
byte plaintext[100] = { }; // data
byte ciphertext[100];

wc_AesXtsSetKey(&aes, key, 32, AES_ENCRYPTION, NULL, INVALID_DEVID);
wc_AesXtsEncryptInit(&aes, tweak, 16, &stream);
int ret = wc_AesXtsEncryptUpdate(&aes, ciphertext, plaintext, 100,
                                 &stream);
if (ret != 0) {
    // encryption failed
}
wc_AesXtsFree(&aes);
```

C.1.2.81 function wc_AesXtsDecryptUpdate

```
int wc_AesXtsDecryptUpdate(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    struct XtsAesStreamData * stream
)
```

This function performs an update step of streaming AES XTS decryption. It processes a chunk of data and can be called multiple times.

Parameters:

- **aes** pointer to the XtsAes structure
- **out** pointer to buffer to store decrypted data
- **in** pointer to ciphertext data to decrypt

- **sz** length of data in bytes
- **stream** pointer to XtsAesStreamData structure for streaming state

See:

- wc_AesXtsDecryptInit
- wc_AesXtsDecryptFinal

Return:

- 0 On success.
- BAD_FUNC_ARG If parameters are invalid.

Example

```
XtsAes aes;
struct XtsAesStreamData stream;
byte key[32] = { }; // 256-bit key
byte tweak[16] = { }; // tweak value
byte ciphertext[100] = { }; // encrypted data
byte plaintext[100];

wc_AesXtsSetKey(&aes, key, 32, AES_DECRYPTION, NULL, INVALID_DEVID);
wc_AesXtsDecryptInit(&aes, tweak, 16, &stream);
int ret = wc_AesXtsDecryptUpdate(&aes, plaintext, ciphertext, 100,
                                 &stream);
if (ret != 0) {
    // decryption failed
}
wc_AesXtsFree(&aes);
```

C.1.2.82 function wc_AesXtsEncryptFinal

```
int wc_AesXtsEncryptFinal(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    struct XtsAesStreamData * stream
)
```

This function finalizes streaming AES XTS encryption. It processes any remaining data and completes the encryption operation.

Parameters:

- **aes** pointer to the XtsAes structure
- **out** pointer to buffer to store final encrypted data
- **in** pointer to final plaintext data to encrypt
- **sz** length of final data in bytes
- **stream** pointer to XtsAesStreamData structure for streaming state

See:

- wc_AesXtsEncryptUpdate
- wc_AesXtsEncryptInit

Return:

- 0 On success.
- BAD_FUNC_ARG If parameters are invalid.

Example

```
XtsAes aes;
struct XtsAesStreamData stream;
byte key[32] = { }; // 256-bit key
byte tweak[16] = { }; // tweak value
byte plaintext[50] = { }; // final data
byte ciphertext[50];

wc_AesXtsSetKey(&aes, key, 32, AES_ENCRYPTION, NULL, INVALID_DEVID);
wc_AesXtsEncryptInit(&aes, tweak, 16, &stream);
// ... update calls ...
int ret = wc_AesXtsEncryptFinal(&aes, ciphertext, plaintext, 50,
                                &stream);
if (ret != 0) {
    // finalization failed
}
wc_AesXtsFree(&aes);
```

C.1.2.83 function wc_AesXtsDecryptFinal

```
int wc_AesXtsDecryptFinal(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    struct XtsAesStreamData * stream
)
```

This function finalizes streaming AES XTS decryption. It processes any remaining data and completes the decryption operation.

Parameters:

- **aes** pointer to the XtsAes structure
- **out** pointer to buffer to store final decrypted data
- **in** pointer to final ciphertext data to decrypt
- **sz** length of final data in bytes
- **stream** pointer to XtsAesStreamData structure for streaming state

See:

- wc_AesXtsDecryptUpdate
- wc_AesXtsDecryptInit

Return:

- 0 On success.
- BAD_FUNC_ARG If parameters are invalid.

Example

```
XtsAes aes;
struct XtsAesStreamData stream;
byte key[32] = { }; // 256-bit key
byte tweak[16] = { }; // tweak value
byte ciphertext[50] = { }; // final encrypted data
byte plaintext[50];
```

```
wc_AesXtsSetKey(&aes, key, 32, AES_DECRYPTION, NULL, INVALID_DEVID);
wc_AesXtsDecryptInit(&aes, tweak, 16, &stream);
// ... update calls ...
int ret = wc_AesXtsDecryptFinal(&aes, plaintext, ciphertext, 50,
                                &stream);
if (ret != 0) {
    // finalization failed
}
wc_AesXtsFree(&aes);
```

C.1.2.84 function wc_AesGetKeySize

```
int wc_AesGetKeySize(
    Aes * aes,
    word32 * keySize
)
```

This function retrieves the key size from an initialized AES structure. It returns the size of the key currently set in the AES object.

Parameters:

- **aes** pointer to the AES structure
- **keySize** pointer to word32 to store the key size in bytes

See:

- wc_AesSetKey
- wc_AesInit

Return:

- 0 On success.
- BAD_FUNC_ARG If aes or keySize is NULL.

Example

```
Aes aes;
byte key[16] = { }; // 128-bit key
word32 keySize;

wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesSetKey(&aes, key, 16, NULL, AES_ENCRYPTION);
int ret = wc_AesGetKeySize(&aes, &keySize);
if (ret == 0) {
    // keySize now contains 16
}
wc_AesFree(&aes);
```

C.1.2.85 function wc_AesInit_Id

```
int wc_AesInit_Id(
    Aes * aes,
    unsigned char * id,
    int len,
    void * heap,
    int devId
)
```

This function initializes an AES structure with an ID. This is useful for tracking or identifying specific AES instances in applications that manage multiple AES contexts.

Parameters:

- **aes** pointer to the AES structure to initialize
- **id** pointer to the ID buffer
- **len** length of the ID in bytes
- **heap** pointer to heap hint for memory allocation (can be NULL)
- **devId** device ID for hardware acceleration (use INVALID_DEVID for software)

See:

- wc_AesInit
- wc_AesInit_Label

Return:

- 0 On success.
- BAD_FUNC_ARG If aes is NULL, or if id is NULL while len is positive.
- BUFFER_E If len is negative or greater than AES_MAX_ID_LEN.

Note: This API is only available when WOLF_PRIVATE_KEY_ID is defined, which is set for PKCS11 support.

Example

```
Aes aes;
byte id[8] = { }; // unique identifier

int ret = wc_AesInit_Id(&aes, id, 8, NULL, INVALID_DEVID);
if (ret != 0) {
    // initialization failed
}
wc_AesFree(&aes);
```

C.1.2.86 function wc_AesInit_Label

```
int wc_AesInit_Label(
    Aes * aes,
    const char * label,
    void * heap,
    int devId
)
```

This function initializes an AES structure with a label string. This is useful for tracking or identifying specific AES instances with human-readable names.

Parameters:

- **aes** pointer to the AES structure to initialize
- **label** pointer to the null-terminated label string
- **heap** pointer to heap hint for memory allocation (can be NULL)
- **devId** device ID for hardware acceleration (use INVALID_DEVID for software)

See:

- wc_AesInit
- wc_AesInit_Id

Return:

- 0 On success.
- BAD_FUNC_ARG If aes or label is NULL.

Note: This API is only available when WOLF_PRIVATE_KEY_ID is defined, which is set for PKCS11 support.

Example

```
Aes aes;

int ret = wc_AesInit_Label(&aes, "MyAESContext", NULL, INVALID_DEVID);
if (ret != 0) {
    // initialization failed
}
wc_AesFree(&aes);
```

C.1.2.87 function wc_AesNew

```
Aes * wc_AesNew(
    void * heap,
    int devId,
    int * result_code
)
```

This function allocates and initializes a new AES structure. It returns a pointer to the allocated structure, which must be freed with wc_AesDelete when no longer needed. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.

Parameters:

- **heap** pointer to heap hint for memory allocation (can be NULL)
- **devId** device ID for hardware acceleration (use INVALID_DEVID for software)
- **result_code** pointer to int to store result code (can be NULL)

See:

- wc_AesDelete
- wc_AesInit

Return:

- Pointer to allocated Aes structure on success.
- NULL on allocation failure.

Note: This API is only available when WC_NO_CONSTRUCTORS is not defined. WC_NO_CONSTRUCTORS is automatically defined when WOLFSSL_NO_MALLOC is defined.

Example

```
int result;
Aes* aes = wc_AesNew(NULL, INVALID_DEVID, &result);
if (aes == NULL || result != 0) {
    // allocation or initialization failed
}
// use aes...
wc_AesDelete(aes, &aes);
```

C.1.2.88 function wc_AesDelete

```
int wc_AesDelete(
    Aes * aes,
    Aes ** aes_p
)
```

This function frees an AES structure that was allocated with wc_AesNew. It also sets the pointer to NULL to prevent use-after-free. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.

Parameters:

- **aes** pointer to the AES structure to free
- **aes_p** pointer to the AES pointer (will be set to NULL)

See:

- wc_AesNew
- wc_AesFree

Return:

- 0 On success.
- BAD_FUNC_ARG If aes or aes_p is NULL.

Note: This API is only available when WC_NO_CONSTRUCTORS is not defined. WC_NO_CONSTRUCTORS is automatically defined when WOLFSSL_NO_MALLOC is defined.

Example

```
Aes* aes = wc_AesNew(NULL, INVALID_DEVID, NULL);
if (aes != NULL) {
    // use aes...
    int ret = wc_AesDelete(aes, &aes);
    // aes is now NULL
}
```

C.1.2.89 function wc_AesSivEncrypt_ex

```
int wc_AesSivEncrypt_ex(
    const byte * key,
    word32 keySz,
    const AesSivAssoc * assoc,
    word32 numAssoc,
    const byte * nonce,
    word32 nonceSz,
    const byte * in,
    word32 inSz,
    byte * siv,
    byte * out
)
```

This function performs AES-SIV (Synthetic IV) encryption with extended parameters. AES-SIV provides nonce-misuse resistance and deterministic authenticated encryption.

Parameters:

- **key** pointer to the key buffer (32, 48, or 64 bytes for SIV)
- **keySz** length of the key in bytes
- **assoc** pointer to array of associated data structures

- **numAssoc** number of associated data items
- **nonce** pointer to the nonce buffer (can be NULL)
- **nonceSz** length of the nonce in bytes
- **in** pointer to plaintext to encrypt
- **inSz** length of plaintext in bytes
- **siv** pointer to buffer to store the SIV (16 bytes)
- **out** pointer to buffer to store ciphertext

See:

- wc_AesSivDecrypt_ex
- wc_AesSivEncrypt

Return:

- 0 On success.
- BAD_FUNC_ARG If parameters are invalid.
- Other negative values on error.

Example

```
byte key[32] = { }; // 256-bit key for AES-128-SIV
AesSivAssoc assoc[1];
byte aad[20] = { }; // associated data
byte nonce[12] = { }; // nonce
byte plaintext[100] = { }; // data
byte siv[16];
byte ciphertext[100];

assoc[0].data = aad;
assoc[0].sz = 20;

int ret = wc_AesSivEncrypt_ex(key, 32, assoc, 1, nonce, 12,
                              plaintext, 100, siv, ciphertext);
if (ret != 0) {
    // encryption failed
}
```

C.1.2.90 function wc_AesSivDecrypt_ex

```
int wc_AesSivDecrypt_ex(
    const byte * key,
    word32 keySz,
    const AesSivAssoc * assoc,
    word32 numAssoc,
    const byte * nonce,
    word32 nonceSz,
    const byte * in,
    word32 inSz,
    byte * siv,
    byte * out
)
```

This function performs AES-SIV (Synthetic IV) decryption with extended parameters. It verifies the SIV and decrypts the ciphertext.

Parameters:

- **key** pointer to the key buffer (32, 48, or 64 bytes for SIV)

- **keySz** length of the key in bytes
- **assoc** pointer to array of associated data structures
- **numAssoc** number of associated data items
- **nonce** pointer to the nonce buffer (can be NULL)
- **nonceSz** length of the nonce in bytes
- **in** pointer to ciphertext to decrypt
- **inSz** length of ciphertext in bytes
- **siv** pointer to the SIV to verify (16 bytes)
- **out** pointer to buffer to store plaintext

See:

- wc_AesSivEncrypt_ex
- wc_AesSivDecrypt

Return:

- 0 On successful decryption and verification.
- AES_SIV_AUTH_E If SIV verification fails.
- BAD_FUNC_ARG If parameters are invalid.
- Other negative values on error.

Example

```
byte key[32] = { }; // 256-bit key for AES-128-SIV
AesSivAssoc assoc[1];
byte aad[20] = { }; // associated data
byte nonce[12] = { }; // nonce
byte ciphertext[100] = { }; // encrypted data
byte siv[16] = { }; // received SIV
byte plaintext[100];

assoc[0].data = aad;
assoc[0].sz = 20;

int ret = wc_AesSivDecrypt_ex(key, 32, assoc, 1, nonce, 12,
                              ciphertext, 100, siv, plaintext);
if (ret != 0) {
    // decryption or verification failed
}
```

C.1.3 Source code

```
int  wc_AesSetKey(Aes* aes, const byte* key, word32 len,
                              const byte* iv, int dir);

int  wc_AesSetIV(Aes* aes, const byte* iv);

int  wc_AesCbcEncrypt(Aes* aes, byte* out,
                                const byte* in, word32 sz);

int  wc_AesCbcDecrypt(Aes* aes, byte* out,
                                const byte* in, word32 sz);

int wc_AesCtrEncrypt(Aes* aes, byte* out,
                                 const byte* in, word32 sz);
```

```
int wc_AesEncryptDirect(Aes* aes, byte* out, const byte* in);

int wc_AesDecryptDirect(Aes* aes, byte* out, const byte* in);

int  wc_AesSetKeyDirect(Aes* aes, const byte* key, word32 len,
                                const byte* iv, int dir);

int  wc_AesGcmSetKey(Aes* aes, const byte* key, word32 len);

int  wc_AesSetTagLen(Aes* aes, word32 tagLen);

int  wc_AesGcmEncrypt(Aes* aes, byte* out,
                                   const byte* in, word32 sz,
                                   const byte* iv, word32 ivSz,
                                   byte* authTag, word32 authTagSz,
                                   const byte* authIn, word32 authInSz);

int  wc_AesGcmDecrypt(Aes* aes, byte* out,
                                   const byte* in, word32 sz,
                                   const byte* iv, word32 ivSz,
                                   const byte* authTag, word32 authTagSz,
                                   const byte* authIn, word32 authInSz);

int wc_GmacSetKey(Gmac* gmac, const byte* key, word32 len);

int wc_GmacUpdate(Gmac* gmac, const byte* iv, word32 ivSz,
                               const byte* authIn, word32 authInSz,
                               byte* authTag, word32 authTagSz);

int  wc_AesCcmSetKey(Aes* aes, const byte* key, word32 keySz);

int  wc_AesCcmEncrypt(Aes* aes, byte* out,
                                   const byte* in, word32 inSz,
                                   const byte* nonce, word32 nonceSz,
                                   byte* authTag, word32 authTagSz,
                                   const byte* authIn, word32 authInSz);

int  wc_AesCcmDecrypt(Aes* aes, byte* out,
                                   const byte* in, word32 inSz,
                                   const byte* nonce, word32 nonceSz,
                                   const byte* authTag, word32 authTagSz,
                                   const byte* authIn, word32 authInSz);

int wc_AesXtsInit(XtsAes* aes, void* heap, int devId);


int wc_AesXtsSetKeyNoInit(XtsAes* aes, const byte* key,
         word32 len, int dir);


int wc_AesXtsSetKey(XtsAes* aes, const byte* key,
         word32 len, int dir, void* heap, int devId);
```

```
int wc_AesXtsEncryptSector(XtsAes* aes, byte* out,
        const byte* in, word32 sz, word64 sector);

int wc_AesXtsDecryptSector(XtsAes* aes, byte* out,
        const byte* in, word32 sz, word64 sector);

int wc_AesXtsEncrypt(XtsAes* aes, byte* out,
        const byte* in, word32 sz, const byte* i, word32 iSz);

int wc_AesXtsDecrypt(XtsAes* aes, byte* out,
       const byte* in, word32 sz, const byte* i, word32 iSz);

int wc_AesXtsFree(XtsAes* aes);


int  wc_AesInit(Aes* aes, void* heap, int devId);

void wc_AesFree(Aes* aes);

int wc_AesCfbEncrypt(Aes* aes, byte* out, const byte* in, word32 sz);

int wc_AesCfbDecrypt(Aes* aes, byte* out, const byte* in, word32 sz);

int wc_AesSivEncrypt(const byte* key, word32 keySz, const byte* assoc,
                     word32 assocSz, const byte* nonce, word32 nonceSz,
                     const byte* in, word32 inSz, byte* siv, byte* out);

int wc_AesSivDecrypt(const byte* key, word32 keySz, const byte* assoc,
                     word32 assocSz, const byte* nonce, word32 nonceSz,
                     const byte* in, word32 inSz, byte* siv, byte* out);




WOLFSSL_API int  wc_AesEaxEncryptAuth(const byte* key, word32 keySz, byte* out,
                                      const byte* in, word32 inSz,
                                      const byte* nonce, word32 nonceSz,
                                      /* output computed auth tag */
                                      byte* authTag, word32 authTagSz,
                                      /* input data to authenticate */
                                      const byte* authIn, word32 authInSz);
WOLFSSL_API int  wc_AesEaxDecryptAuth(const byte* key, word32 keySz, byte* out,
                                      const byte* in, word32 inSz,
                                      const byte* nonce, word32 nonceSz,
                                      /* auth tag to verify against */
                                      const byte* authTag, word32 authTagSz,
                                      /* input data to authenticate */
                                      const byte* authIn, word32 authInSz);

WOLFSSL_API int  wc_AesEaxInit(AesEax* eax,
                               const byte* key, word32 keySz,
```

```
                                const byte* nonce, word32 nonceSz,
                                const byte* authIn, word32 authInSz);

WOLFSSL_API int  wc_AesEaxEncryptUpdate(AesEax* eax, byte* out,
                                        const byte* in, word32 inSz,
                                        const byte* authIn, word32 authInSz);

WOLFSSL_API int  wc_AesEaxDecryptUpdate(AesEax* eax, byte* out,
                                        const byte* in, word32 inSz,
                                        const byte* authIn, word32 authInSz);
WOLFSSL_API int  wc_AesEaxAuthDataUpdate(AesEax* eax,
                                       const byte* authIn, word32 authInSz);

WOLFSSL_API int wc_AesEaxEncryptFinal(AesEax* eax,
                                      byte* authTag, word32 authTagSz);

WOLFSSL_API int wc_AesEaxDecryptFinal(AesEax* eax,
                                      const byte* authIn, word32 authInSz);
WOLFSSL_API int wc_AesEaxFree(AesEax* eax);

int wc_AesCtsEncrypt(const byte* key, word32 keySz, byte* out,
                     const byte* in, word32 inSz,
                     const byte* iv);

int wc_AesCtsEncrypt(const byte* key, word32 keySz, byte* out,
                     const byte* in, word32 inSz,
                     const byte* iv);

int wc_AesCtsDecrypt(const byte* key, word32 keySz, byte* out,
                     const byte* in, word32 inSz,
                     const byte* iv);

int wc_AesCtsEncryptUpdate(Aes* aes, byte* out, word32* outSz,
                           const byte* in, word32 inSz);

int wc_AesCtsEncryptFinal(Aes* aes, byte* out, word32* outSz);

int wc_AesCtsDecryptUpdate(Aes* aes, byte* out, word32* outSz,
                           const byte* in, word32 inSz);

int wc_AesCtsDecryptFinal(Aes* aes, byte* out, word32* outSz);


int wc_AesCfb1Encrypt(Aes* aes, byte* out, const byte* in, word32 sz);

int wc_AesCfb8Encrypt(Aes* aes, byte* out, const byte* in, word32 sz);

int wc_AesCfb1Decrypt(Aes* aes, byte* out, const byte* in, word32 sz);

int wc_AesCfb8Decrypt(Aes* aes, byte* out, const byte* in, word32 sz);

int wc_AesOfbEncrypt(Aes* aes, byte* out, const byte* in, word32 sz);

int wc_AesOfbDecrypt(Aes* aes, byte* out, const byte* in, word32 sz);
```

```
int wc_AesEcbEncrypt(Aes* aes, byte* out, const byte* in, word32 sz);

int wc_AesEcbDecrypt(Aes* aes, byte* out, const byte* in, word32 sz);

int wc_AesCtrSetKey(Aes* aes, const byte* key, word32 len, const byte* iv,
                    int dir);

int wc_AesGcmSetKey_ex(Aes* aes, const byte* key, word32 len, word32 kup);

int wc_AesGcmInit(Aes* aes, const byte* key, word32 len, const byte* iv,
                  word32 ivSz);

int wc_AesGcmEncryptInit(Aes* aes, const byte* key, word32 len,
                         const byte* iv, word32 ivSz);

int wc_AesGcmEncryptInit_ex(Aes* aes, const byte* key, word32 len,
                            byte* ivOut, word32 ivOutSz);

int wc_AesGcmEncryptUpdate(Aes* aes, byte* out, const byte* in, word32 sz,
                           const byte* authIn, word32 authInSz);

int wc_AesGcmEncryptFinal(Aes* aes, byte* authTag, word32 authTagSz);

int wc_AesGcmDecryptInit(Aes* aes, const byte* key, word32 len,
                         const byte* iv, word32 ivSz);

int wc_AesGcmDecryptUpdate(Aes* aes, byte* out, const byte* in, word32 sz,
                           const byte* authIn, word32 authInSz);

int wc_AesGcmDecryptFinal(Aes* aes, const byte* authTag, word32 authTagSz);

int wc_AesGcmSetExtIV(Aes* aes, const byte* iv, word32 ivSz);

int wc_AesGcmSetIV(Aes* aes, word32 ivSz, const byte* ivFixed,
                   word32 ivFixedSz, WC_RNG* rng);

int wc_AesGcmEncrypt_ex(Aes* aes, byte* out, const byte* in, word32 sz,
                        byte* ivOut, word32 ivOutSz, byte* authTag,
                        word32 authTagSz, const byte* authIn,
                        word32 authInSz);

int wc_Gmac(const byte* key, word32 keySz, byte* iv, word32 ivSz,
            const byte* authIn, word32 authInSz, byte* authTag,
            word32 authTagSz, WC_RNG* rng);

int wc_GmacVerify(const byte* key, word32 keySz, const byte* iv,
                  word32 ivSz, const byte* authIn, word32 authInSz,
                  const byte* authTag, word32 authTagSz);

int wc_AesCcmSetNonce(Aes* aes, const byte* nonce, word32 nonceSz);

int wc_AesCcmEncrypt_ex(Aes* aes, byte* out, const byte* in, word32 sz,
                        byte* ivOut, word32 ivOutSz, byte* authTag,
```

```
                        word32 authTagSz, const byte* authIn,
                        word32 authInSz);

int wc_AesKeyWrap(const byte* key, word32 keySz, const byte* in,
                  word32 inSz, byte* out, word32 outSz, const byte* iv);

int wc_AesKeyWrap_ex(Aes *aes, const byte* in, word32 inSz, byte* out,
                     word32 outSz, const byte* iv);

int wc_AesKeyUnWrap(const byte* key, word32 keySz, const byte* in,
                    word32 inSz, byte* out, word32 outSz, const byte* iv);

int wc_AesKeyUnWrap_ex(Aes *aes, const byte* in, word32 inSz, byte* out,
                       word32 outSz, const byte* iv);

int wc_AesXtsEncryptConsecutiveSectors(XtsAes* aes, byte* out,
                                       const byte* in, word32 sz,
                                       word64 sector, word32 sectorSz);

int wc_AesXtsDecryptConsecutiveSectors(XtsAes* aes, byte* out,
                                       const byte* in, word32 sz,
                                       word64 sector, word32 sectorSz);

int wc_AesXtsEncryptInit(XtsAes* aes, const byte* i, word32 iSz,
                         struct XtsAesStreamData *stream);

int wc_AesXtsDecryptInit(XtsAes* aes, const byte* i, word32 iSz,
                         struct XtsAesStreamData *stream);

int wc_AesXtsEncryptUpdate(XtsAes* aes, byte* out, const byte* in,
                           word32 sz, struct XtsAesStreamData *stream);

int wc_AesXtsDecryptUpdate(XtsAes* aes, byte* out, const byte* in,
                           word32 sz, struct XtsAesStreamData *stream);

int wc_AesXtsEncryptFinal(XtsAes* aes, byte* out, const byte* in,
                          word32 sz, struct XtsAesStreamData *stream);

int wc_AesXtsDecryptFinal(XtsAes* aes, byte* out, const byte* in,
                          word32 sz, struct XtsAesStreamData *stream);

int wc_AesGetKeySize(Aes* aes, word32* keySize);

int wc_AesInit_Id(Aes* aes, unsigned char* id, int len, void* heap,
                  int devId);

int wc_AesInit_Label(Aes* aes, const char* label, void* heap, int devId);

Aes* wc_AesNew(void* heap, int devId, int *result_code);

int wc_AesDelete(Aes* aes, Aes** aes_p);

int wc_AesSivEncrypt_ex(const byte* key, word32 keySz,
                        const AesSivAssoc* assoc, word32 numAssoc,
```

```
                        const byte* nonce, word32 nonceSz, const byte* in,
                        word32 inSz, byte* siv, byte* out);

int wc_AesSivDecrypt_ex(const byte* key, word32 keySz,
                        const AesSivAssoc* assoc, word32 numAssoc,
                        const byte* nonce, word32 nonceSz, const byte* in,
                        word32 inSz, byte* siv, byte* out);
```

C.2 arc4.h

C.2.1 Functions

	Name
int	**wc_Arc4Process**(Arc4 * arc4, byte * out, const byte * in, word32 length)This function encrypts an input message from the buffer in, placing the ciphertext in the output buffer out, or decrypts a ciphertext from the buffer in, placing the plaintext in the output buffer out, using ARC4 encryption. This function is used for both encryption and decryption. Before this method may be called, one must first initialize the ARC4 structure using wc_Arc4SetKey.
int	**wc_Arc4SetKey**(Arc4 * arc4, const byte * key, word32 length)This function sets the key for a ARC4 object, initializing it for use as a cipher. It should be called before using it for encryption with wc_Arc4Process.
int	**wc_Arc4Init**(Arc4 * arc4, void * heap, int devId)This function initializes an ARC4 structure for use with asynchronous cryptographic operations. It sets up the heap hint and device ID for hardware acceleration support.
void	**wc_Arc4Free**(Arc4 * arc4)This function frees an ARC4 structure, releasing any resources allocated for asynchronous cryptographic operations. It should be called when the ARC4 structure is no longer needed.

C.2.2 Functions Documentation

C.2.2.1 function wc_Arc4Process

```
int wc_Arc4Process(
    Arc4 * arc4,
    byte * out,
    const byte * in,
    word32 length
)
```

This function encrypts an input message from the buffer in, placing the ciphertext in the output buffer out, or decrypts a ciphertext from the buffer in, placing the plaintext in the output buffer out, using

ARC4 encryption. This function is used for both encryption and decryption. Before this method may be called, one must first initialize the ARC4 structure using wc_Arc4SetKey.

Parameters:

- **arc4** pointer to the ARC4 structure used to process the message
- **out** pointer to the output buffer in which to store the processed message
- **in** pointer to the input buffer containing the message to process
- **length** length of the message to process

See: wc_Arc4SetKey

Return:

- 0 Returned upon successfully processing the message.
- BAD_FUNC_ARG If arc4, out, or in is NULL.
- MISSING_KEY If no key has been set on the ARC4 structure with wc_Arc4SetKey.

Example

```
Arc4 enc;
byte key[] = { key to use for encryption };
wc_Arc4SetKey(&enc, key, sizeof(key));

byte plain[] = { plain text to encode };
byte cipher[sizeof(plain)];
byte decrypted[sizeof(plain)];
// encrypt the plain into cipher
wc_Arc4Process(&enc, cipher, plain, sizeof(plain));
// decrypt the cipher
wc_Arc4Process(&enc, decrypted, cipher, sizeof(cipher));
```

C.2.2.2 function wc_Arc4SetKey

```
int wc_Arc4SetKey(
    Arc4 * arc4,
    const byte * key,
    word32 length
)
```

This function sets the key for a ARC4 object, initializing it for use as a cipher. It should be called before using it for encryption with wc_Arc4Process.

Parameters:

- **arc4** pointer to an arc4 structure to be used for encryption
- **key** key with which to initialize the arc4 structure
- **length** length of the key used to initialize the arc4 structure

See: wc_Arc4Process

Return: none

Example

```
Arc4 enc;
byte key[] = { initialize with key to use for encryption };
wc_Arc4SetKey(&enc, key, sizeof(key));
```

C.2.2.3 function wc_Arc4Init

```
int wc_Arc4Init(
    Arc4 * arc4,
    void * heap,
    int devId
)
```

This function initializes an ARC4 structure for use with asynchronous cryptographic operations. It sets up the heap hint and device ID for hardware acceleration support.

Parameters:

- **arc4** pointer to the Arc4 structure to initialize
- **heap** pointer to heap hint for memory allocation (can be NULL)
- **devId** device ID for hardware acceleration (use INVALID_DEVID for software)

See:

- wc_Arc4SetKey
- wc_Arc4Free

Return:

- 0 On success.
- BAD_FUNC_ARG If arc4 is NULL.

Example

```
Arc4 arc4;
int ret = wc_Arc4Init(&arc4, NULL, INVALID_DEVID);
if (ret != 0) {
    // initialization failed
}
// use arc4 for encryption/decryption
wc_Arc4Free(&arc4);
```

C.2.2.4 function wc_Arc4Free

```
void wc_Arc4Free(
    Arc4 * arc4
)
```

This function frees an ARC4 structure, releasing any resources allocated for asynchronous cryptographic operations. It should be called when the ARC4 structure is no longer needed.

Parameters:

- **arc4** pointer to the Arc4 structure to free

See:

- wc_Arc4Init
- wc_Arc4SetKey

Return: none No return value.

Example

```
Arc4 arc4;
wc_Arc4Init(&arc4, NULL, INVALID_DEVID);
wc_Arc4SetKey(&arc4, key, keyLen);
```

```
// use arc4 for encryption/decryption
wc_Arc4Free(&arc4);
```

C.2.3 Source code

```
int wc_Arc4Process(Arc4* arc4, byte* out, const byte* in, word32 length);

int wc_Arc4SetKey(Arc4* arc4, const byte* key, word32 length);

int wc_Arc4Init(Arc4* arc4, void* heap, int devId);

void wc_Arc4Free(Arc4* arc4);
```

C.3 ascon.h

C.3.1 Functions

	Name
int	**wc_AsconHash256_Init**(wc_AsconHash256 * a)This function initializes the ASCON context for hashing.
int	**wc_AsconHash256_Update**(wc_AsconHash256 * a, const byte * data, word32 dataSz)This function updates the ASCON hash with the input data.
int	**wc_AsconHash256_Final**(wc_AsconHash256 * a, byte * hash)This function finalizes the ASCON hash and produces the output. Resets state of the ASCON hash context.
wc_AsconAEAD128 *	**wc_AsconAEAD128_New**(void)This function allocates and initializes a new Ascon AEAD context.
void	**wc_AsconAEAD128_Free**(wc_AsconAEAD128 * a)This function frees the resources associated with the Ascon AEAD context.
int	**wc_AsconAEAD128_Init**(wc_AsconAEAD128 * a)This function initializes an Ascon AEAD context.
void	**wc_AsconAEAD128_Clear**(wc_AsconAEAD128 * a)This function deinitializes an Ascon AEAD context. It does not free the context.
int	**wc_AsconAEAD128_SetKey**(wc_AsconAEAD128 * a, const byte * key)This function sets the key for the Ascon AEAD context.
int	**wc_AsconAEAD128_SetNonce**(wc_AsconAEAD128 * a, const byte * nonce)This function sets the nonce for the Ascon AEAD context.
int	**wc_AsconAEAD128_SetAD**(wc_AsconAEAD128 * a, const byte * ad, word32 adSz)This function sets the associated data for the Ascon AEAD context.

	Name
int	**wc_AsconAEAD128_EncryptUpdate**(wc_AsconAEAD128 * a, byte * out, const byte * in, word32 inSz)This function encrypts a plaintext message using Ascon AEAD. The output is stored in the out buffer. The length of the output is equal to the length of the input.
int	**wc_AsconAEAD128_EncryptFinal**(wc_AsconAEAD128 * a, byte * tag)This function finalizes the encryption process using Ascon AEAD and produces the authentication tag. Resets state of the Ascon AEAD context.
int	**wc_AsconAEAD128_DecryptUpdate**(wc_AsconAEAD128 * a, byte * out, const byte * in, word32 inSz)This function updates the decryption process using Ascon AEAD. The output is stored in the out buffer. The length of the output is equal to the length of the input.
int	**wc_AsconAEAD128_DecryptFinal**(wc_AsconAEAD128 * a, const byte * tag)This function finalizes the decryption process using Ascon AEAD and verifies the authentication tag. Resets state of the Ascon AEAD context.
wc_AsconHash256 *	**wc_AsconHash256_New**(void)This function allocates and initializes a new Ascon Hash256 context. The returned context must be freed with wc_AsconHash256_Free when no longer needed.
void	**wc_AsconHash256_Free**(wc_AsconHash256 * a)This function frees an Ascon Hash256 context that was allocated with wc_AsconHash256_New. It clears the context before freeing to prevent information leakage.
void	**wc_AsconHash256_Clear**(wc_AsconHash256 * a)This function clears an Ascon Hash256 context by zeroing all internal state. This should be called to securely erase sensitive data from memory.

C.3.2 Functions Documentation

C.3.2.1 function wc_AsconHash256_Init

```
int wc_AsconHash256_Init(
    wc_AsconHash256 * a
)
```

This function initializes the ASCON context for hashing.

Parameters:

- **a** pointer to the ASCON context to initialize.

See:

- wc_AsconHash256_Update

- wc_AsconHash256_Final

Return:

- 0 on success.
- BAD_FUNC_ARG if the context pointer is NULL.

Example

```
wc_AsconHash256 a;
byte data[] = {0x01, 0x02, 0x03};
byte hash[ASCON_HASH256_SZ];

if (wc_AsconHash256_Init(&a) != 0)
    // handle error
if (wc_AsconHash256_Update(&ctx, data, sizeof(data)) != 0)
    // handle error
if (wc_AsconHash256_Final(&ctx, hash, sizeof(hash)) != 0)
    // handle error
// hash contains the final hash
```

C.3.2.2 function wc_AsconHash256_Update

```
int wc_AsconHash256_Update(
    wc_AsconHash256 * a,
    const byte * data,
    word32 dataSz
)
```

This function updates the ASCON hash with the input data.

Parameters:

- **ctx** pointer to the ASCON context.
- **in** pointer to the input data.
- **inSz** size of the input data.

See:

- wc_AsconHash256_Init
- wc_AsconHash256_Final

Return:

- 0 on success.
- BAD_FUNC_ARG if the context or input pointer is NULL.

Example

```
wc_AsconHash256 a;
byte data[] = {0x01, 0x02, 0x03};
byte hash[ASCON_HASH256_SZ];

if (wc_AsconHash256_Init(&a) != 0)
    // handle error
if (wc_AsconHash256_Update(&ctx, data, sizeof(data)) != 0)
    // handle error
if (wc_AsconHash256_Final(&ctx, hash, sizeof(hash)) != 0)
    // handle error
// hash contains the final hash
```

C.3.2.3 function wc_AsconHash256_Final

```
int wc_AsconHash256_Final(
    wc_AsconHash256 * a,
    byte * hash
)
```

This function finalizes the ASCON hash and produces the output. Resets state of the ASCON hash context.

Parameters:

- **ctx** pointer to the ASCON context.
- **out** pointer to the output buffer.
- **outSz** size of the output buffer, should be at least ASCON_HASH256_SZ.

See:

- wc_AsconHash256_Init
- wc_AsconHash256_Update

Return:

- 0 on success.
- BAD_FUNC_ARG if the context or output pointer is NULL.

Example

```
wc_AsconHash256 a;
byte data[] = {0x01, 0x02, 0x03};
byte hash[ASCON_HASH256_SZ];

if (wc_AsconHash256_Init(&a) != 0)
    // handle error
if (wc_AsconHash256_Update(&ctx, data, sizeof(data)) != 0)
    // handle error
if (wc_AsconHash256_Final(&ctx, hash, sizeof(hash)) != 0)
    // handle error
// hash contains the final hash
```

C.3.2.4 function wc_AsconAEAD128_New

```
wc_AsconAEAD128 * wc_AsconAEAD128_New(
    void
)
```

This function allocates and initializes a new Ascon AEAD context.

See:

- wc_AsconAEAD128_Free
- wc_AsconAEAD128_Free
- wc_AsconAEAD128_Init

Return:

- pointer to the newly allocated Ascon AEAD context
- NULL on failure.
- Pointer to allocated wc_AsconAEAD128 structure on success.
- NULL on allocation or initialization failure.

This function allocates and initializes a new Ascon AEAD128 context. The returned context must be freed with wc_AsconAEAD128_Free when no longer needed.

Example

```
wc_AsconAEAD128* a = wc_AsconAEAD128_New();
if (a == NULL) {
    // handle allocation error
}
wc_AsconAEAD128_Free(a);
```

Example

```
wc_AsconAEAD128* aead = wc_AsconAEAD128_New();
if (aead == NULL) {
    // handle allocation error
}
byte key[ASCON_AEAD128_KEY_SZ] = { }; // key
byte nonce[ASCON_AEAD128_NONCE_SZ] = { }; // nonce
wc_AsconAEAD128_SetKey(aead, key);
wc_AsconAEAD128_SetNonce(aead, nonce);
// perform encryption/decryption
wc_AsconAEAD128_Free(aead);
```

C.3.2.5 function wc_AsconAEAD128_Free

```
void wc_AsconAEAD128_Free(
    wc_AsconAEAD128 * a
)
```

This function frees the resources associated with the Ascon AEAD context.

Parameters:

- **a** pointer to the Ascon AEAD context to free.

See: wc_AsconAEAD128_New

Example

```
wc_AsconAEAD128* a = wc_AsconAEAD128_New();
if (a == NULL) {
    // handle allocation error
}
// Use the context
wc_AsconAEAD128_Free(a);
```

C.3.2.6 function wc_AsconAEAD128_Init

```
int wc_AsconAEAD128_Init(
    wc_AsconAEAD128 * a
)
```

This function initializes an Ascon AEAD context.

Parameters:

- **a** pointer to the Ascon AEAD context to initialize.

See:

- wc_AsconAeadEncrypt

- wc_AsconAeadDecrypt

Return:

- 0 on success.
- BAD_FUNC_ARG if the context or output pointer is NULL.

Example

```
AsconAead a;

if (wc_AsconAEAD128_Init(&a) != 0)
    // handle error
```

C.3.2.7 function wc_AsconAEAD128_Clear

```
void wc_AsconAEAD128_Clear(
    wc_AsconAEAD128 * a
)
```

This function deinitializes an Ascon AEAD context. It does not free the context.

Parameters:

- **a** pointer to the Ascon AEAD context to deinitialize.

See:

- wc_AsconAeadEncrypt
- wc_AsconAeadDecrypt

Example

```
AsconAead a;

if (wc_AsconAEAD128_Init(&a) != 0)
    // handle error
wc_AsconAEAD128_Clear(&a);
```

C.3.2.8 function wc_AsconAEAD128_SetKey

```
int wc_AsconAEAD128_SetKey(
    wc_AsconAEAD128 * a,
    const byte * key
)
```

This function sets the key for the Ascon AEAD context.

Parameters:

- **a** pointer to the initialized Ascon AEAD context.
- **key** pointer to the key buffer of length ASCON_AEAD128_KEY_SZ.

See:

- wc_AsconAEAD128_Init
- wc_AsconAEAD128_SetNonce
- wc_AsconAEAD128_SetAD

Return:

- 0 on success.
- BAD_FUNC_ARG if the context or key pointer is NULL.

- BAD_STATE_E if the key has already been set.

Example

```
wc_AsconAEAD128 a;
byte key[ASCON_AEAD128_KEY_SZ] = { ... };

if (wc_AsconAEAD128_Init(&a) != 0)
    // handle error
if (wc_AsconAEAD128_SetKey(&a, key) != 0)
    // handle error
```

C.3.2.9 function wc_AsconAEAD128_SetNonce

```
int wc_AsconAEAD128_SetNonce(
    wc_AsconAEAD128 * a,
    const byte * nonce
)
```

This function sets the nonce for the Ascon AEAD context.

Parameters:

- **a** pointer to the initialized Ascon AEAD context.
- **nonce** pointer to the nonce buffer of length ASCON_AEAD128_NONCE_SZ.

See:

- wc_AsconAEAD128_Init
- wc_AsconAEAD128_SetKey
- wc_AsconAEAD128_SetAD

Return:

- 0 on success.
- BAD_FUNC_ARG if the context or nonce pointer is NULL.
- BAD_STATE_E if the nonce has already been set.

Example

```
wc_AsconAEAD128 a;
byte nonce[ASCON_AEAD128_NONCE_SZ] = { ... };

if (wc_AsconAEAD128_Init(&a) != 0)
    // handle error
if (wc_AsconAEAD128_SetNonce(&a, nonce) != 0)
    // handle error
```

C.3.2.10 function wc_AsconAEAD128_SetAD

```
int wc_AsconAEAD128_SetAD(
    wc_AsconAEAD128 * a,
    const byte * ad,
    word32 adSz
)
```

This function sets the associated data for the Ascon AEAD context.

Parameters:

- **a** pointer to the initialized Ascon AEAD context.

- **ad** pointer to the associated data buffer.
- **adSz** size of the associated data buffer.

See:

- wc_AsconAEAD128_Init
- wc_AsconAEAD128_SetKey
- wc_AsconAEAD128_SetNonce

Return:

- 0 on success.
- BAD_FUNC_ARG if the context is NULL, or if the associated data pointer is NULL while adSz is greater than 0. A NULL associated data pointer is permitted when adSz is 0.
- BAD_STATE_E if the key or nonce has not been set, or if the associated data has already been set. The associated data must be provided in a single call.

Example

```
wc_AsconAEAD128 a;
byte key[ASCON_AEAD128_KEY_SZ] = { ... };
byte nonce[ASCON_AEAD128_NONCE_SZ] = { ... };
byte ad[] = { ... };

if (wc_AsconAEAD128_Init(&a) != 0)
    // handle error
if (wc_AsconAEAD128_SetKey(&a, key) != 0)
    // handle error
if (wc_AsconAEAD128_SetNonce(&a, nonce) != 0)
    // handle error
if (wc_AsconAEAD128_SetAD(&a, ad, sizeof(ad)) != 0)
    // handle error
```

C.3.2.11 function wc_AsconAEAD128_EncryptUpdate

```
int wc_AsconAEAD128_EncryptUpdate(
    wc_AsconAEAD128 * a,
    byte * out,
    const byte * in,
    word32 inSz
)
```

This function encrypts a plaintext message using Ascon AEAD. The output is stored in the out buffer. The length of the output is equal to the length of the input.

Parameters:

- **a** pointer to the initialized Ascon AEAD context.
- **out** pointer to the output buffer to store the ciphertext.
- **in** pointer to the input buffer containing the plaintext message.
- **inSz** length of the input buffer.

See:

- wc_AsconAeadInit
- wc_AsconAEAD128_Clear
- wc_AsconAEAD128_SetKey
- wc_AsconAEAD128_SetNonce
- wc_AsconAEAD128_SetAD
- wc_AsconAEAD128_EncryptFinal

- wc_AsconAEAD128_DecryptUpdate
- wc_AsconAEAD128_DecryptFinal

Return:

- 0 on success.
- BAD_FUNC_ARG if the context or output pointer is NULL or the input is NULL while the input size is greater than 0.
- BAD_STATE_E if the key, nonce, or additional data has not been set or the context was previously used for decryption.

Example

```
wc_AsconAEAD128 a;
byte key[ASCON_AEAD128_KEY_SZ] = { ... };
byte nonce[ASCON_AEAD128_NONCE_SZ] = { ... };
byte plaintext[PLAIN_TEXT_SIZE] = { ... };
byte ciphertext[CIPHER_TEXT_SIZE];
byte tag[ASCON_AEAD128_TAG_SZ] = { ... };

if (wc_AsconAeadInit(&a) != 0)
    // handle error
if (wc_AsconAEAD128_SetKey(&a, key) != 0)
    // handle error
if (wc_AsconAEAD128_SetNonce(&a, nonce) != 0)
    // handle error
if (wc_AsconAEAD128_SetAD(&a, ad, sizeof(ad)) != 0)
    // handle error
if (wc_AsconAEAD128_EncryptUpdate(&a, ciphertext, plaintext,
                                  sizeof(plaintext)) != 0)
    // handle error
if (wc_AsconAEAD128_EncryptFinal(&a, tag) != 0)
    // handle error
```

C.3.2.12 function wc_AsconAEAD128_EncryptFinal

```
int wc_AsconAEAD128_EncryptFinal(
    wc_AsconAEAD128 * a,
    byte * tag
)
```

This function finalizes the encryption process using Ascon AEAD and produces the authentication tag. Resets state of the Ascon AEAD context.

Parameters:

- **a** pointer to the initialized Ascon AEAD context.
- **tag** pointer to the output buffer to store the authentication tag.

See:

- wc_AsconAEAD128_Init
- wc_AsconAEAD128_SetKey
- wc_AsconAEAD128_SetNonce
- wc_AsconAEAD128_SetAD
- wc_AsconAEAD128_EncryptUpdate
- wc_AsconAEAD128_DecryptUpdate
- wc_AsconAEAD128_DecryptFinal

Return:

- 0 on success.
- BAD_FUNC_ARG if the context or output pointer is NULL or the input is NULL while the input size is greater than 0.
- BAD_STATE_E if the key, nonce, or additional data has not been set or the context was previously used for decryption.

Example

```
wc_AsconAEAD128 a;
byte key[ASCON_AEAD128_KEY_SZ] = { ... };
byte nonce[ASCON_AEAD128_NONCE_SZ] = { ... };
byte plaintext[PLAIN_TEXT_SIZE] = { ... };
byte ciphertext[CIPHER_TEXT_SIZE];
byte tag[ASCON_AEAD128_TAG_SZ] = { ... };

if (wc_AsconAeadInit(&a) != 0)
    // handle error
if (wc_AsconAEAD128_SetKey(&a, key) != 0)
    // handle error
if (wc_AsconAEAD128_SetNonce(&a, nonce) != 0)
    // handle error
if (wc_AsconAEAD128_SetAD(&a, ad, sizeof(ad)) != 0)
    // handle error
if (wc_AsconAEAD128_EncryptUpdate(&a, ciphertext, plaintext,
                                  sizeof(plaintext)) != 0)
    // handle error
if (wc_AsconAEAD128_EncryptFinal(&a, tag) != 0)
    // handle error
```

C.3.2.13 function wc_AsconAEAD128_DecryptUpdate

```
int wc_AsconAEAD128_DecryptUpdate(
    wc_AsconAEAD128 * a,
    byte * out,
    const byte * in,
    word32 inSz
)
```

This function updates the decryption process using Ascon AEAD. The output is stored in the out buffer. The length of the output is equal to the length of the input.

Parameters:

- **a** pointer to the initialized Ascon AEAD context.
- **out** pointer to the output buffer to store the plaintext.
- **in** pointer to the input buffer containing the ciphertext message.
- **inSz** length of the input buffer.

See:

- wc_AsconAEAD128_Init
- wc_AsconAEAD128_SetKey
- wc_AsconAEAD128_SetNonce
- wc_AsconAEAD128_SetAD
- wc_AsconAEAD128_EncryptUpdate
- wc_AsconAEAD128_EncryptFinal

- wc_AsconAEAD128_DecryptFinal

Return:

- 0 on success.
- BAD_FUNC_ARG if the context or output pointer is NULL or the input is NULL while the input size is greater than 0.
- BAD_STATE_E if the key, nonce, or additional data has not been set or the context was previously used for encryption.

Example

```
wc_AsconAEAD128 a;
byte key[ASCON_AEAD128_KEY_SZ] = { ... };
byte nonce[ASCON_AEAD128_NONCE_SZ] = { ... };
byte ciphertext[CIPHER_TEXT_SIZE] = { ... };
byte plaintext[PLAIN_TEXT_SIZE];
byte tag[ASCON_AEAD128_TAG_SZ] = { ... };

if (wc_AsconAeadInit(&a) != 0)
    // handle error
if (wc_AsconAEAD128_SetKey(&a, key) != 0)
    // handle error
if (wc_AsconAEAD128_SetNonce(&a, nonce) != 0)
    // handle error
if (wc_AsconAEAD128_SetAD(&a, ad, sizeof(ad)) != 0)
    // handle error
if (wc_AsconAEAD128_DecryptUpdate(&a, plaintext, ciphertext,
                                  sizeof(ciphertext)) != 0)
    // handle error
if (wc_AsconAEAD128_DecryptFinal(&a, tag) != 0)
    // handle error
```

C.3.2.14 function wc_AsconAEAD128_DecryptFinal

```
int wc_AsconAEAD128_DecryptFinal(
    wc_AsconAEAD128 * a,
    const byte * tag
)
```

This function finalizes the decryption process using Ascon AEAD and verifies the authentication tag. Resets state of the Ascon AEAD context.

Parameters:

- **a** pointer to the initialized Ascon AEAD context.
- **tag** pointer to the buffer containing the authentication tag to verify

See:

- wc_AsconAEAD128_Init
- wc_AsconAEAD128_SetKey
- wc_AsconAEAD128_SetNonce
- wc_AsconAEAD128_SetAD
- wc_AsconAEAD128_DecryptUpdate
- wc_AsconAEAD128_EncryptUpdate
- wc_AsconAEAD128_EncryptFinal

Return:

- 0 on success.
- BAD_FUNC_ARG if the context or tag pointer is NULL.
- BAD_STATE_E if the key, nonce, or additional data has not been set or the context was previously used for encryption.
- ASCON_AUTH_E if the authentication tag does not match.

Example

```
wc_AsconAEAD128 a;
byte key[ASCON_AEAD128_KEY_SZ] = { ... };
byte nonce[ASCON_AEAD128_NONCE_SZ] = { ... };
byte ciphertext[CIPHER_TEXT_SIZE] = { ... };
byte plaintext[PLAIN_TEXT_SIZE];
byte tag[ASCON_AEAD128_TAG_SZ] = { ... };

if (wc_AsconAeadInit(&a) != 0)
    // handle error
if (wc_AsconAEAD128_SetKey(&a, key) != 0)
    // handle error
if (wc_AsconAEAD128_SetNonce(&a, nonce) != 0)
    // handle error
if (wc_AsconAEAD128_SetAD(&a, ad, sizeof(ad)) != 0)
    // handle error
if (wc_AsconAEAD128_DecryptUpdate(&a, plaintext, ciphertext,
                                  sizeof(ciphertext)) != 0)
    // handle error
if (wc_AsconAEAD128_DecryptFinal(&a, tag) != 0)
    // handle error
```

C.3.2.15 function wc_AsconHash256_New

```
wc_AsconHash256 * wc_AsconHash256_New(
    void
)
```

This function allocates and initializes a new Ascon Hash256 context. The returned context must be freed with wc_AsconHash256_Free when no longer needed.

See:

- wc_AsconHash256_Free
- wc_AsconHash256_Init

Return:

- Pointer to allocated wc_AsconHash256 structure on success.
- NULL on allocation or initialization failure.

Example

```
wc_AsconHash256* hash = wc_AsconHash256_New();
if (hash == NULL) {
    // handle allocation error
}
byte data[]; // data to hash
wc_AsconHash256_Update(hash, data, sizeof(data));
byte digest[ASCON_HASH256_SZ];
wc_AsconHash256_Final(hash, digest);
wc_AsconHash256_Free(hash);
```

C.3.2.16 function wc_AsconHash256_Free

```
void wc_AsconHash256_Free(
    wc_AsconHash256 * a
)
```

This function frees an Ascon Hash256 context that was allocated with wc_AsconHash256_New. It clears the context before freeing to prevent information leakage.

Parameters:

- **a** pointer to the wc_AsconHash256 structure to free

See:

- wc_AsconHash256_New
- wc_AsconHash256_Clear

Return: none No return value.

Example

```
wc_AsconHash256* hash = wc_AsconHash256_New();
if (hash != NULL) {
    // use hash context
    wc_AsconHash256_Free(hash);
}
```

C.3.2.17 function wc_AsconHash256_Clear

```
void wc_AsconHash256_Clear(
    wc_AsconHash256 * a
)
```

This function clears an Ascon Hash256 context by zeroing all internal state. This should be called to securely erase sensitive data from memory.

Parameters:

- **a** pointer to the wc_AsconHash256 structure to clear

See:

- wc_AsconHash256_Init
- wc_AsconHash256_Free

Return: none No return value.

Example

```
wc_AsconHash256 hash;
wc_AsconHash256_Init(&hash);
byte data[]; // data to hash
wc_AsconHash256_Update(&hash, data, sizeof(data));
byte digest[ASCON_HASH256_SZ];
wc_AsconHash256_Final(&hash, digest);
wc_AsconHash256_Clear(&hash);
```

C.3.3 Source code

```
int wc_AsconHash256_Init(wc_AsconHash256* a);
```

```
int wc_AsconHash256_Update(wc_AsconHash256* a, const byte* data, word32
 ↪  dataSz);

int wc_AsconHash256_Final(wc_AsconHash256* a, byte* hash);

wc_AsconAEAD128* wc_AsconAEAD128_New(void);

void wc_AsconAEAD128_Free(wc_AsconAEAD128 *a);


int wc_AsconAEAD128_Init(wc_AsconAEAD128* a);

void wc_AsconAEAD128_Clear(wc_AsconAEAD128 *a);

int wc_AsconAEAD128_SetKey(wc_AsconAEAD128* a, const byte* key);

int wc_AsconAEAD128_SetNonce(wc_AsconAEAD128* a, const byte* nonce);

int wc_AsconAEAD128_SetAD(wc_AsconAEAD128* a, const byte* ad, word32 adSz);

int wc_AsconAEAD128_EncryptUpdate(wc_AsconAEAD128* a, byte* out, const byte*
 ↪  in,
                                   word32 inSz);

int wc_AsconAEAD128_EncryptFinal(wc_AsconAEAD128* a, byte* tag);

int wc_AsconAEAD128_DecryptUpdate(wc_AsconAEAD128* a, byte* out, const byte*
 ↪  in,
                                   word32 inSz);

int wc_AsconAEAD128_DecryptFinal(wc_AsconAEAD128* a, const byte* tag);

wc_AsconHash256* wc_AsconHash256_New(void);

void wc_AsconHash256_Free(wc_AsconHash256* a);

void wc_AsconHash256_Clear(wc_AsconHash256* a);

wc_AsconAEAD128* wc_AsconAEAD128_New(void);
```

C.4 asn.h

C.4.1 Functions

	Name
int	**wc_BerToDer**(const byte * ber, word32 berSz, byte * der, word32 * derSz)This function converts BER (Basic Encoding Rules) formatted data to DER (Distinguished Encoding Rules) format. BER allows indefinite length encoding while DER requires definite lengths. This function calculates definite lengths for all indefinite length items.
void	**FreeAltNames**(DNS_entry * altNames, void * heap)This function frees a linked list of alternative names (DNS_entry structures). It deallocates each node and its associated name string, IP string, and RID string if present.
int	**wc_SetUnknownExtCallbackEx**(DecodedCert * cert, wc_UnknownExtCallbackEx cb, void * ctx)This function sets an extended callback for handling unknown certificate extensions during certificate parsing. The callback receives additional context information compared to the basic callback.
int	**wc_CheckCertSignature**(const byte * cert, word32 certSz, void * heap, WOLFSSL_CERT_MANAGER * cm)This function verifies the signature on a certificate using a certificate manager. It checks that the certificate is properly signed by a trusted CA.
int	**wc_EncodeObjectId**(const word16 * in, word32 inSz, byte * out, word32 * outSz)This function encodes an array of word16 values into an ASN.1 Object Identifier (OID) in DER format. OIDs are used to identify algorithms, extensions, and other objects in certificates and cryptographic protocols.
word32	**SetAlgoID**(int algoOID, byte * output, int type, int curveSz)This function sets the algorithm identifier in DER format. It encodes the algorithm OID and optional parameters based on the algorithm type and curve size.
int	**wc_DhPublicKeyDecode**(const byte * input, word32 * inOutIdx, DhKey * key, word32 inSz)This function decodes a DER encoded Diffie-Hellman public key. It extracts the public key value from the DER encoding and stores it in the DhKey structure.
int	**wc_SignCert_cb**(int requestSz, int sType, byte * buf, word32 buffSz, int keyType, wc_SignCertCb signCb, void * signCtx, WC_RNG * rng)Sign a certificate or CSR using a callback function.

C.4.2 Functions Documentation

C.4.2.1 function wc_BerToDer

```
int wc_BerToDer(
    const byte * ber,
    word32 berSz,
    byte * der,
    word32 * derSz
)
```

This function converts BER (Basic Encoding Rules) formatted data to DER (Distinguished Encoding Rules) format. BER allows indefinite length encoding while DER requires definite lengths. This function calculates definite lengths for all indefinite length items.

Parameters:

- **ber** pointer to the buffer containing BER formatted data
- **berSz** size of the BER data in bytes
- **der** pointer to buffer to store DER formatted data (can be NULL to calculate required size)
- **derSz** pointer to size of der buffer; updated with actual size needed or used

See: wc_EncodeObjectId

Return:

- 0 On success.
- ASN_PARSE_E If the BER data is invalid.
- BAD_FUNC_ARG If ber or derSz are NULL.
- BUFFER_E If der is not NULL and derSz is too small.

Note: This API is not public by default. Define WOLFSSL_PUBLIC_ASN to expose APIs marked WOLFSSL_ASN_API.

Example

```
byte ber[256] = { }; // BER encoded data
byte der[256];
word32 derSz = sizeof(der);

int ret = wc_BerToDer(ber, sizeof(ber), der, &derSz);
if (ret == 0) {
    // der now contains DER formatted data of length derSz
}
```

C.4.2.2 function FreeAltNames

```
void FreeAltNames(
    DNS_entry * altNames,
    void * heap
)
```

This function frees a linked list of alternative names (DNS_entry structures). It deallocates each node and its associated name string, IP string, and RID string if present.

Parameters:

- **altNames** pointer to the head of the alternative names linked list
- **heap** pointer to heap hint for memory deallocation (can be NULL)

See: AltNameNew

Return: none No return value.

Note: This API is not public by default. Define WOLFSSL_PUBLIC_ASN to expose APIs marked WOLFSSL_ASN_API.

Example

```
DNS_entry* altNames = NULL;
// populate altNames with certificate alternative names

FreeAltNames(altNames, NULL);
// altNames list is now freed
```

C.4.2.3 function wc_SetUnknownExtCallbackEx

```
int wc_SetUnknownExtCallbackEx(
    DecodedCert * cert,
    wc_UnknownExtCallbackEx cb,
    void * ctx
)
```

This function sets an extended callback for handling unknown certificate extensions during certificate parsing. The callback receives additional context information compared to the basic callback.

Parameters:

- **cert** pointer to the DecodedCert structure
- **cb** callback function to handle unknown extensions
- **ctx** context pointer passed to the callback

See:

- wc_SetUnknownExtCallback
- wc_InitDecodedCert

Return:

- 0 On success.
- BAD_FUNC_ARG If cert is NULL.

Note: This API is not public by default. Define WOLFSSL_PUBLIC_ASN to expose APIs marked WOLFSSL_ASN_API.

Example

```
DecodedCert cert;

int UnknownExtCallback(const byte* oid, word32 oidSz, int crit,
                       const byte* der, word32 derSz, void* ctx) {
    // handle unknown extension
    return 0;
}

wc_InitDecodedCert(&cert, derCert, derCertSz, NULL);
wc_SetUnknownExtCallbackEx(&cert, UnknownExtCallback, myContext);
wc_ParseCert(&cert, CERT_TYPE, NO_VERIFY, NULL);
```

C.4.2.4 function wc_CheckCertSignature

```
int wc_CheckCertSignature(
    const byte * cert,
    word32 certSz,
    void * heap,
    WOLFSSL_CERT_MANAGER * cm
)
```

This function verifies the signature on a certificate using a certificate manager. It checks that the certificate is properly signed by a trusted CA.

Parameters:

- **cert** pointer to the DER encoded certificate
- **certSz** size of the certificate in bytes
- **heap** pointer to heap hint for memory allocation (can be NULL)
- **cm** pointer to certificate manager containing trusted CAs

See:

- wolfSSL_CertManagerNew
- wolfSSL_CertManagerLoadCA

Return:

- 0 On successful signature verification.
- ASN_SIG_CONFIRM_E If signature verification fails.
- Other negative values on error.

Example

```
byte cert[2048] = { }; // DER encoded certificate
word32 certSz = sizeof(cert);
WOLFSSL_CERT_MANAGER* cm;

cm = wolfSSL_CertManagerNew();
wolfSSL_CertManagerLoadCA(cm, "ca-cert.pem", NULL);

int ret = wc_CheckCertSignature(cert, certSz, NULL, cm);
if (ret == 0) {
    // certificate signature is valid
}
wolfSSL_CertManagerFree(cm);
```

C.4.2.5 function wc_EncodeObjectId

```
int wc_EncodeObjectId(
    const word16 * in,
    word32 inSz,
    byte * out,
    word32 * outSz
)
```

This function encodes an array of word16 values into an ASN.1 Object Identifier (OID) in DER format. OIDs are used to identify algorithms, extensions, and other objects in certificates and cryptographic protocols.

Parameters:

- **in** pointer to array of word16 values representing OID components
- **inSz** number of components in the OID
- **out** pointer to buffer to store encoded OID (can be NULL to calculate size)
- **outSz** pointer to size of out buffer; updated with actual size

See: wc_BerToDer

Return:

- 0 On success.

- BAD_FUNC_ARG If in, inSz, or outSz are invalid.
- BUFFER_E If out is not NULL and outSz is too small.

Example

```
word16 oid[] = {1, 2, 840, 113549, 1, 1, 11}; // sha256WithRSAEncryption
byte encoded[32];
word32 encodedSz = sizeof(encoded);

int ret = wc_EncodeObjectId(oid, sizeof(oid)/sizeof(word16),
                            encoded, &encodedSz);
if (ret == 0) {
    // encoded contains DER encoded OID
}
```

C.4.2.6 function SetAlgoID

```
word32 SetAlgoID(
    int algoOID,
    byte * output,
    int type,
    int curveSz
)
```

This function sets the algorithm identifier in DER format. It encodes the algorithm OID and optional parameters based on the algorithm type and curve size.

Parameters:

- **algoOID** algorithm object identifier constant
- **output** pointer to buffer to store encoded algorithm ID
- **type** type of encoding (oidSigType, oidHashType, etc.)
- **curveSz** size of the curve for ECC algorithms (0 for non-ECC)

See: wc_EncodeObjectId

Return:

- Length of the encoded algorithm identifier on success.
- Negative value on error.

Example

```
byte algId[32];
word32 len;

len = SetAlgoID(CTC_SHA256wRSA, algId, oidSigType, 0);
if (len > 0) {
    // algId contains encoded algorithm identifier
}
```

C.4.2.7 function wc_DhPublicKeyDecode

```
int wc_DhPublicKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    DhKey * key,
    word32 inSz
)
```

This function decodes a DER encoded Diffie-Hellman public key. It extracts the public key value from the DER encoding and stores it in the DhKey structure.

Parameters:

- **input** pointer to buffer containing DER encoded public key
- **inOutIdx** pointer to index in buffer; updated to end of key
- **key** pointer to DhKey structure to store decoded public key
- **inSz** size of the input buffer

See:

- wc_InitDhKey
- wc_DhKeyDecode

Return:

- 0 On success.
- BAD_FUNC_ARG If input, inOutIdx, key, or inSz are invalid.
- ASN_PARSE_E If the DER encoding is invalid.
- Other negative values on error.

Example

```
byte derKey[256] = { }; // DER encoded DH public key
word32 idx = 0;
DhKey key;

wc_InitDhKey(&key);
int ret = wc_DhPublicKeyDecode(derKey, &idx, &key, sizeof(derKey));
if (ret == 0) {
    // key now contains the decoded public key
}
wc_FreeDhKey(&key);
```

C.4.2.8 function wc_SignCert_cb

```
int wc_SignCert_cb(
    int requestSz,
    int sType,
    byte * buf,
    word32 buffSz,
    int keyType,
    wc_SignCertCb signCb,
    void * signCtx,
    WC_RNG * rng
)
```

Sign a certificate or CSR using a callback function.

Parameters:

- **requestSz** Size of the certificate body to sign (from Cert.bodySz).
- **sType** Signature algorithm type (e.g., CTC_SHA256wRSA, CTC_SHA256wECDSA).
- **buf** Buffer containing the certificate/CSR DER data to sign.
- **buffSz** Total size of the buffer (must be large enough for signature).
- **keyType** Type of key used for signing. Only RSA_TYPE and ECC_TYPE are supported.
- **signCb** User-provided signing callback function.
- **signCtx** Context pointer passed to the signing callback.
- **rng** Random number generator (may be NULL if not needed).

See:

- wc_SignCertCb
- wc_SignCert
- wc_SignCert_ex
- wc_MakeCert
- wc_MakeCertReq

Return:

- Size of the signed certificate/CSR on success.
- BAD_FUNC_ARG if signCb or buf is NULL, buffSz is 0, or keyType is not RSA_TYPE or ECC_TYPE.
- BUFFER_E if the buffer is too small for the signed certificate.
- MEMORY_E if memory allocation fails.
- Negative error code on other failures.

This function signs a certificate or Certificate Signing Request (CSR) using a user-provided signing callback. This allows external signing implementations (e.g., TPM, HSM) without requiring the crypto callback infrastructure, making it suitable for FIPS-compliant applications.

The function performs the following:

1. Hashes the certificate/CSR body according to the signature algorithm
2. Encodes the hash (RSA) or prepares it for signing (ECC)
3. Calls the user-provided callback to perform the actual signing
4. Encodes the signature into the certificate/CSR DER structure

NOTE: Only RSA and ECC key types are supported. Ed25519, Ed448, and post-quantum algorithms (Falcon, Dilithium, SLH-DSA) sign messages directly rather than hashes, so they cannot use this callback-based API. Use wc_SignCert_ex for those algorithms.

NOTE: This function does NOT support async crypto (WOLFSSL_ASYNC_CRYPT). The internal context is local to this function and cannot persist across async re-entry.

Example

```
Cert cert;
byte derBuf[4096];
int derSz;
MySignCtx myCtx;

wc_InitCert(&cert);

derSz = wc_MakeCert(&cert, derBuf, sizeof(derBuf), NULL, NULL, &rng);

derSz = wc_SignCert_cb(cert.bodySz, cert.sigType, derBuf, sizeof(derBuf),
                       RSA_TYPE, mySignCallback, &myCtx, &rng);
if (derSz > 0) {
    printf("Signed certificate is %d bytes\n", derSz);
}
```

C.4.3 Source code

```
int wc_BerToDer(const byte* ber, word32 berSz, byte* der, word32* derSz);

void FreeAltNames(DNS_entry* altNames, void* heap);

int wc_SetUnknownExtCallbackEx(DecodedCert* cert,
```

```
                                wc_UnknownExtCallbackEx cb, void *ctx);

int wc_CheckCertSignature(const byte* cert, word32 certSz, void* heap,
                          WOLFSSL_CERT_MANAGER* cm);

int wc_EncodeObjectId(const word16* in, word32 inSz, byte* out,
                      word32* outSz);

word32 SetAlgoID(int algoOID, byte* output, int type, int curveSz);

int wc_DhPublicKeyDecode(const byte* input, word32* inOutIdx, DhKey* key,
                         word32 inSz);

int wc_SignCert_cb(int requestSz, int sType, byte* buf, word32 buffSz,
                   int keyType, wc_SignCertCb signCb, void* signCtx,
                   WC_RNG* rng);
```

C.5 asn_public.h

C.5.1 Functions

	Name
int	**wc_InitCert**(Cert * cert)This function initializes a default cert, with the default options: version = 3 (0x2), serial = 0, sigType = SHA_WITH_RSA, issuer = blank, daysValid = 500, selfSigned = 1 (true) use subject as issuer, subject = blank.
Cert *	**wc_CertNew**(void * heap)This function allocates a new Cert structure for use during cert operations without the application having to allocate the structure itself. The Cert structure is also initialized by this function thus removing the need to call wc_InitCert(). When the application is finished using the allocated Cert structure wc_CertFree() must be called.
int	**wc_InitCert_ex**(Cert * cert, void * heap, int devId)Initializes certificate with heap hint and device ID.
void	**wc_CertFree**(Cert * cert)This function frees the memory allocated for a cert structure by a previous call to wc_CertNew().
int	**wc_MakeCert**(Cert * cert, byte * derBuffer, word32 derSz, RsaKey * rsaKey, ecc_key * eccKey, WC_RNG * rng)Used to make CA signed certs. Called after the subject information has been entered. This function makes an x509 Certificate v3 RSA or ECC from a cert input. It then writes this cert to derBuffer. It takes in either an rsaKey or an eccKey to generate the certificate. The certificate must be initialized with wc_InitCert before this method is called.

	Name
int	**wc_MakeCert_ex**(Cert * cert, byte * derBuffer, word32 derSz, int keyType, void * key, WC_RNG * rng)Makes certificate with generic key type support.
int	**wc_MakeCertReq_ex**(Cert * cert, byte * derBuffer, word32 derSz, int keyType, void * key)Makes certificate request with generic key type support.
int	**wc_SignCert_ex**(int requestSz, int sType, byte * buf, word32 buffSz, int keyType, void * key, WC_RNG * rng)Signs certificate with generic key type support.
int	**wc_MakeSigWithBitStr**(byte * sig, int sigSz, int sType, byte * buf, word32 bufSz, int keyType, void * key, WC_RNG * rng)Makes signature with bit string encoding. This function is used for dual algorithm certificate signing, where an alternative signature is created using a secondary key algorithm (e.g., a post_quantum algorithm alongside a traditional algorithm).
int	**wc_GetCertDates**(Cert * cert, struct tm * before, struct tm * after)Gets certificate validity dates.
int	**wc_GetDateInfo**(const byte * certDate, int certDateSz, const byte ** date, byte * format, int * length)Extracts date information from certificate date field. This function parses an ASN.1 encoded date (including tag and length) and returns a pointer to the raw date value bytes, the ASN.1 time type, and the length of the date value.
int	**wc_GetDateAsCalendarTime**(const byte * date, int length, byte format, struct tm * timearg)Converts certificate date to calendar time structure.
int	**wc_MakeCertReq**(Cert * cert, byte * derBuffer, word32 derSz, RsaKey * rsaKey, ecc_key * eccKey)This function makes a certificate signing request using the input certificate and writes the output to derBuffer. It takes in either an rsaKey or an eccKey to generate the certificate request. wc_SignCert() will need to be called after this function to sign the certificate request. Please see the wolfCrypt test application (./wolfcrypt/test/test.c) for an example usage of this function.

	Name
int	**wc_SignCert**(int requestSz, int sigType, byte * derBuffer, word32 derSz, RsaKey * rsaKey, ecc_key * eccKey, WC_RNG * rng)This function signs buffer and adds the signature to the end of buffer. It takes in a signature type. Must be called after wc_MakeCert() or wc_MakeCertReq() if creating a CA signed cert.
int	**wc_MakeSelfCert**(Cert * cert, byte * derBuffer, word32 derSz, RsaKey * key, WC_RNG * rng)This function is a combination of the previous two functions, wc_MakeCert and wc_SignCert for self signing (the previous functions may be used for CA requests). It makes a certificate, and then signs it, generating a self-signed certificate.
int	**wc_SetIssuer**(Cert * cert, const char * issuerFile)This function sets the issuer for a certificate to the issuer in the provided pem issuerFile. It also changes the certificate's self-signed attribute to false. The issuer specified in issuerFile is verified prior to setting the cert issuer. This method is used to set fields prior to signing.
int	**wc_SetSubject**(Cert * cert, const char * subjectFile)This function sets the subject for a certificate to the subject in the provided pem subjectFile. This method is used to set fields prior to signing.
int	**wc_SetSubjectRaw**(Cert * cert, const byte * der, int derSz)This function sets the raw subject for a certificate from the subject in the provided der buffer. This method is used to set the raw subject field prior to signing.
int	**wc_GetSubjectRaw**(byte ** subjectRaw, Cert * cert)This function gets the raw subject from the certificate structure.
int	**wc_SetAltNames**(Cert * cert, const char * file)This function sets the alternate names for a certificate to the alternate names in the provided pem file. This is useful in the case that one wishes to secure multiple domains with the same certificate. This method is used to set fields prior to signing.
int	**wc_SetIssuerBuffer**(Cert * cert, const byte * der, int derSz)This function sets the issuer for a certificate from the issuer in the provided der buffer. It also changes the certificate's self-signed attribute to false. This method is used to set fields prior to signing.

	Name
int	**wc_SetIssuerRaw**(Cert * cert, const byte * der, int derSz)This function sets the raw issuer for a certificate from the issuer in the provided der buffer. This method is used to set the raw issuer field prior to signing.
int	**wc_SetSubjectBuffer**(Cert * cert, const byte * der, int derSz)This function sets the subject for a certificate from the subject in the provided der buffer. This method is used to set fields prior to signing.
int	**wc_SetAltNamesBuffer**(Cert * cert, const byte * der, int derSz)This function sets the alternate names for a certificate from the alternate names in the provided der buffer. This is useful in the case that one wishes to secure multiple domains with the same certificate. This method is used to set fields prior to signing.
int	**wc_SetDNSEntry**(void * heap, const char * str, int strLen, int type, DNS_entry ** entries)This function allocates a single subject alternative name (SAN) entry, copies the supplied name into it, sets its GeneralName type and length, and appends it to the linked list pointed to by entries. The name is duplicated internally, so the caller's buffer need not outlive the call. The resulting list can be encoded with wc_FlattenAltNames and must be freed with FreeAltNames.
int	**wc_FlattenAltNames**(byte * output, word32 outputSz, const DNS_entry * names)This function encodes a linked list of subject alternative name entries into the DER GeneralNames SEQUENCE used as the value of the subjectAltName certificate extension. The output is suitable for assigning to Cert.altNames (with the return value stored in Cert.altNamesSz) prior to signing.
int	**wc_SetAltNamesFromList**(Cert * cert, const DNS_entry * names)This function encodes a linked list of subject alternative name entries directly into a Cert structure, ready for signing. It is a convenience wrapper around wc_FlattenAltNames: the list is encoded into cert->altNames and the encoded length is stored in cert->altNamesSz, so the caller does not have to manage the buffer or size bookkeeping. The supplied list is not consumed and must still be freed by the caller with FreeAltNames.

	Name
int	**wc_SetDatesBuffer**(Cert * cert, const byte * der, int derSz)This function sets the dates for a certificate from the date range in the provided der buffer. This method is used to set fields prior to signing.
int	**wc_SetAuthKeyIdFromPublicKey**(Cert * cert, RsaKey * rsakey, ecc_key * eckey)Set AKID from either an RSA or ECC public key. note: Only set one of rsakey or eckey, not both.
int	**wc_SetAuthKeyIdFromPublicKey_ex**(Cert * cert, int keyType, void * key)Sets authority key ID from public key with generic key type.
int	**wc_SetAuthKeyIdFromCert**(Cert * cert, const byte * der, int derSz)Set AKID from from DER encoded certificate.
int	**wc_SetAuthKeyId**(Cert * cert, const char * file)Set AKID from certificate file in PEM format.
int	**wc_SetSubjectKeyIdFromPublicKey**(Cert * cert, RsaKey * rsakey, ecc_key * eckey)Set SKID from RSA or ECC public key.
int	**wc_SetSubjectKeyIdFromPublicKey_ex**(Cert * cert, int keyType, void * key)Sets subject key ID from public key with generic key type.
int	**wc_SetSubjectKeyId**(Cert * cert, const char * file)Set SKID from public key file in PEM format. Both arguments are required.
int	**wc_SetKeyUsage**(Cert * cert, const char * value)This function allows you to set the key usage using a comma delimited string of tokens. Accepted tokens are: digitalSignature, nonRepudiation, contentCommitment, keyCertSign, cRLSign, dataEncipherment, keyAgreement, keyEncipherment, encipherOnly, decipherOnly. Example: “digitalSignature,nonRepudiation” nonRepudiation and contentCommitment are for the same usage.
int	**wc_SetExtKeyUsage**(Cert * cert, const char * value)Sets extended key usage using comma-delimited string.
int	**wc_SetExtKeyUsageOID**(Cert * cert, const char * oid, word32 sz, byte idx, void * heap)Sets extended key usage using OID string.
int	**wc_PemPubKeyToDer**(const char * fileName, unsigned char * derBuf, int derSz)Loads a PEM key from a file and converts to a DER encoded buffer.
int	**wc_PemPubKeyToDer_ex**(const char * fileName, DerBuffer ** der)Loads PEM public key from file to DER buffer.

	Name
int	**wc_PubKeyPemToDer**(const unsigned char * pem, int pemSz, unsigned char * buff, int buffSz)Convert a PEM encoded public key to DER. Returns the number of bytes written to the buffer or a negative value for an error.
int	**wc_PemGetHeaderFooter**(int type, const char ** header, const char ** footer)Gets PEM header and footer strings for given type.
int	**wc_AllocDer**(DerBuffer ** pDer, word32 length, int type, void * heap)Allocates DER buffer with specified length and type.
void	**wc_FreeDer**(DerBuffer ** pDer)Frees DER buffer allocated by wc_AllocDer or wc_PemToDer.
int	**wc_PemToDer**(const unsigned char * buff, long longSz, int type, DerBuffer ** pDer, void * heap, EncryptedInfo * info, int * keyFormat)Converts PEM to DER format with encryption info support.
int	**wc_PemCertToDer**(const char * fileName, unsigned char * derBuf, int derSz)This function converts a pem certificate to a der certificate, and places the resulting certificate in the derBuf buffer provided.
int	**wc_DerToPem**(const byte * der, word32 derSz, byte * output, word32 outSz, int type)This function converts a der formatted input certificate, contained in the der buffer, into a pem formatted output certificate, contained in the output buffer. It should be noted that this is not an in place conversion, and a separate buffer must be utilized to store the pem formatted output.
int	**wc_DerToPemEx**(const byte * der, word32 derSz, byte * output, word32 outSz, byte * cipher_info, int type)This function converts a der formatted input certificate, contained in the der buffer, into a pem formatted output certificate, contained in the output buffer. It should be noted that this is not an in place conversion, and a separate buffer must be utilized to store the pem formatted output. Allows setting cipher info.
int	**wc_KeyPemToDer**(const unsigned char * pem, int pemSz, unsigned char * buff, int buffSz, const char * pass)Converts a key in PEM format to DER format.
int	**wc_CertPemToDer**(const unsigned char * pem, int pemSz, unsigned char * buff, int buffSz, int type)This function converts a PEM formatted certificate to DER format. Calls OpenSSL function PemToDer.

	Name
int	**wc_PemCertToDer_ex**(const char * fileName, DerBuffer ** der)Loads PEM certificate from file to DER buffer.
word32	**wc_PkcsPad**(byte * buf, word32 sz, word32 blockSz)Adds PKCS padding to buffer for RSA encryption.
int	**wc_RsaPublicKeyDecode_ex**(const byte * input, word32 * inOutIdx, word32 inSz, const byte ** n, word32 * nSz, const byte ** e, word32 * eSz)Decodes RSA public key and extracts modulus and exponent.
int	**wc_RsaPublicKeyDerSize**(RsaKey * key, int with_header)Calculates DER encoded RSA public key size.
int	**wc_RsaPrivateKeyValidate**(const byte * input, word32 * inOutIdx, int * keySz, word32 inSz)Validates DER encoded RSA private key format. This function validates the ASN.1 syntax and structure of the RSA private key (sequences, integer tags, and lengths) without loading the key values into an RsaKey structure. It does not perform mathematical validation of the RSA key parameters (e.g., checking if p and q are prime, or if the key components satisfy RSA mathematical relationships).
int	**wc_GetPubKeyDerFromCert**(struct DecodedCert * cert, byte * derKey, word32 * derKeySz)This function gets the public key in DER format from a populated DecodedCert struct. Users must call wc_InitDecodedCert() and wc_ParseCert() before calling this API. wc_InitDecodedCert() accepts a DER/ASN.1 encoded certificate. To convert a PEM cert to DER, first use wc_CertPemToDer() before calling wc_InitDecodedCert().
int	**wc_DsaParamsDecode**(const byte * input, word32 * inOutIdx, DsaKey * key, word32 inSz)Decodes DSA parameters from DER format.
int	**wc_DsaKeyToParamsDer**(DsaKey * key, byte * output, word32 outLen)Encodes DSA parameters to DER format.
int	**wc_DsaKeyToParamsDer_ex**(DsaKey * key, byte * output, word32 * outLen)Encodes DSA parameters to DER with size output.
int	**wc_DhParamsToDer**(DhKey * key, byte * out, word32 * outSz)Encodes DH parameters to DER format.
int	**wc_DhPubKeyToDer**(DhKey * key, byte * out, word32 * outSz)Encodes DH public key to DER format.

	Name
int	**wc_DhPrivKeyToDer**(DhKey * key, byte * out, word32 * outSz)Encodes DH private key to DER format.
int	**wc_EccPrivateKeyDecode**(const byte * input, word32 * inOutIdx, ecc_key * key, word32 inSz)This function reads in an ECC private key from the input buffer, input, parses the private key, and uses it to generate an ecc_key object, which it stores in key.
int	**wc_EccPrivateKeyToDer**(ecc_key * key, byte * output, word32 outLen)Encodes ECC private key to DER format.
int	**wc_EccKeyDerSize**(ecc_key * key, int pub)Calculates DER encoded ECC key size.
int	**wc_EccPrivateKeyToPKCS8**(ecc_key * key, byte * output, word32 * outLen)Encodes ECC private key to PKCS#8 format.
int	**wc_EccKeyToPKCS8**(ecc_key * key, byte * output, word32 * outLen)Encodes ECC key pair to PKCS#8 format.
int	**wc_EccPublicKeyDerSize**(ecc_key * key, int with_AlgCurve)Calculates DER encoded ECC public key size.
int	**wc_EccKeyToDer**(ecc_key * key, byte * output, word32 outLen)This function writes a private ECC key to der format.
int	**wc_EccPublicKeyDecode**(const byte * input, word32 * inOutIdx, ecc_key * key, word32 inSz)Decodes an ECC public key from an input buffer. It will parse an ASN sequence to retrieve the ECC key.
int	**wc_EccPublicKeyToDer**(ecc_key * key, byte * output, word32 outLen, int with_AlgCurve)This function converts the ECC public key to DER format. It returns the size of buffer used. The public ECC key in DER format is stored in output buffer. The with_AlgCurve flag will include a header that has the Algorithm and Curve information.
int	**wc_EccPublicKeyToDer_ex**(ecc_key * key, byte * output, word32 outLen, int with_AlgCurve, int comp)This function converts the ECC public key to DER format. It returns the size of buffer used. The public ECC key in DER format is stored in output buffer. The with_AlgCurve flag will include a header that has the Algorithm and Curve information. The comp parameter determines if the public key will be exported as compressed.

	Name
int	**wc_Curve25519PrivateKeyDecode**(const byte * input, word32 * inOutIdx, curve25519_key * key, word32 inSz)This function decodes a Curve25519 private key (only) from a DER encoded buffer.
int	**wc_Curve25519PublicKeyDecode**(const byte * input, word32 * inOutIdx, curve25519_key * key, word32 inSz)This function decodes a Curve25519 public key (only) from a DER encoded buffer.
int	**wc_Curve25519KeyDecode**(const byte * input, word32 * inOutIdx, curve25519_key * key, word32 inSz)This function decodes a Curve25519 key from a DER encoded buffer. It can decode either a private key, a public key, or both.
int	**wc_Curve25519PrivateKeyToDer**(curve25519_key * key, byte * output, word32 outLen)This function encodes a Curve25519 private key to DER format. If the input key structure contains a public key, it will be ignored.
int	**wc_Curve25519PublicKeyToDer**(curve25519_key * key, byte * output, word32 outLen, int withAlg)This function encodes a Curve25519 public key to DER format. If the input key structure contains a private key, it will be ignored.
int	**wc_Curve25519KeyToDer**(curve25519_key * key, byte * output, word32 outLen, int withAlg)This function encodes a Curve25519 key to DER format. It can encode either a private key, a public key, or both.
int	**wc_Ed25519PrivateKeyDecode**(const byte * input, word32 * inOutIdx, ed25519_key * key, word32 inSz)Decodes Ed25519 private key from DER format.
int	**wc_Ed25519PrivateKeyDecode_ex**(const byte * input, word32 * inOutIdx, ed25519_key * key, word32 inSz, int trusted)Decodes Ed25519 private key from DER format with control over validation of a bundled public key. When the DER contains a public key and trusted is 0, the public key is validated with wc_ed25519_check_key(), including a check that it matches the private key. When trusted is 1 all validation of the bundled public key is skipped, so trusted should only be set with known_good key material. wc_Ed25519PrivateKeyDecode() behaves as trusted set to 0.

	Name
int	**wc_Ed25519PublicKeyDecode**(const byte * input, word32 * inOutIdx, ed25519_key * key, word32 inSz)Decodes Ed25519 public key from DER format.
int	**wc_Ed25519KeyToDer**(const ed25519_key * key, byte * output, word32 outLen)Encodes Ed25519 key to DER format.
int	**wc_Ed25519PrivateKeyToDer**(const ed25519_key * key, byte * output, word32 outLen)Encodes Ed25519 private key to DER format.
int	**wc_Ed25519PublicKeyToDer**(const ed25519_key * key, byte * output, word32 outLen, int withAlg)Encodes Ed25519 public key to DER format.
int	**wc_Ed448PrivateKeyDecode**(const byte * input, word32 * inOutIdx, ed448_key * key, word32 inSz)Decodes Ed448 private key from DER format.
int	**wc_Ed448PrivateKeyDecode_ex**(const byte * input, word32 * inOutIdx, ed448_key * key, word32 inSz, int trusted)Decodes Ed448 private key from DER format with control over validation of a bundled public key. When the DER contains a public key and trusted is 0, the public key is validated with wc_ed448_check_key(), including a check that it matches the private key. When trusted is 1 all validation of the bundled public key is skipped, so trusted should only be set with known_good key material. wc_Ed448PrivateKeyDecode() behaves as trusted set to 0.
int	**wc_Ed448PublicKeyDecode**(const byte * input, word32 * inOutIdx, ed448_key * key, word32 inSz)Decodes Ed448 public key from DER format.
int	**wc_Ed448KeyToDer**(const ed448_key * key, byte * output, word32 outLen)Encodes Ed448 key to DER format.
int	**wc_Ed448PrivateKeyToDer**(const ed448_key * key, byte * output, word32 outLen)Encodes Ed448 private key to DER format.
int	**wc_Ed448PublicKeyToDer**(const ed448_key * key, byte * output, word32 outLen, int withAlg)Encodes Ed448 public key to DER format.
int	**wc_Curve448PrivateKeyDecode**(const byte * input, word32 * inOutIdx, curve448_key * key, word32 inSz)Decodes Curve448 private key from DER format.

	Name
int	**wc_Curve448PublicKeyDecode**(const byte * input, word32 * inOutIdx, curve448_key * key, word32 inSz)Decodes Curve448 public key from DER format.
int	**wc_Curve448PrivateKeyToDer**(curve448_key * key, byte * output, word32 outLen)Encodes Curve448 private key to DER format.
int	**wc_Curve448PublicKeyToDer**(curve448_key * key, byte * output, word32 outLen)Encodes Curve448 public key to DER format.
word32	**wc_EncodeSignature**(byte * out, const byte * digest, word32 digSz, int hashOID)This function encodes a digital signature into the output buffer, and returns the size of the encoded signature created.
int	**wc_GetCTC_HashOID**(int type)This function returns the hash OID that corresponds to a hashing type. For example, when given the type: WC_SHA512, this function returns the identifier corresponding to a SHA512 hash, SHA512h.
void	**wc_SetCert_Free**(Cert * cert)This function cleans up memory and resources used by the certificate structure's decoded cert cache. When WOLFSSL_CERT_GEN_CACHE is defined the decoded cert structure is cached in the certificate structure. This allows subsequent calls to certificate set functions to avoid parsing the decoded cert on each call.
int	**wc_GetPkcs8TraditionalOffset**(byte * input, word32 * inOutIdx, word32 sz)This function finds the beginning of the traditional private key inside a PKCS#8 unencrypted buffer.
int	**wc_CreatePKCS8Key**(byte * out, word32 * outSz, byte * key, word32 keySz, int algoID, const byte * curveOID, word32 oidSz)This function takes in a DER private key and converts it to PKCS#8 format. Also used in creating PKCS#12 shrouded key bags. See RFC 5208.
int	**wc_EncryptPKCS8Key**(byte * key, word32 keySz, byte * out, word32 * outSz, const char * password, int passwordSz, int vPKCS, int pbeOid, int encAlgId, byte * salt, word32 saltSz, int itt, WC_RNG * rng, void * heap)This function takes in an unencrypted PKCS#8 DER key (e.g. one created by wc_CreatePKCS8Key) and converts it to PKCS#8 encrypted format. The resulting encrypted key can be decrypted using wc_DecryptPKCS8Key. See RFC 5208.

	Name
int	**wc_EncryptPKCS8Key_ex**(byte * key, word32 keySz, byte * out, word32 * outSz, const char * password, int passwordSz, int vPKCS, int pbeOid, int encAlgId, byte * salt, word32 saltSz, int itt, WC_RNG * rng, void * heap, int devId)Encrypts PKCS#8 key with extended parameters.
int	**wc_GetTime**(void * timePtr, word32 timeSize)Gets current time for certificate operations.
int	**wc_EncryptedInfoGet**(EncryptedInfo * info, const char * cipherName)Gets encryption info from encrypted PEM.
int	**wc_ParseCertPIV**(wc_CertPIV * cert, const byte * buf, word32 totalSz)Parses PIV certificate format.
int	**wc_GetSubjectPubKeyInfoDerFromCert**(const byte * certDer, word32 certDerSz, byte * pubKeyDer, word32 * pubKeyDerSz)Extracts subject public key info from certificate.
int	**wc_GetDecodedCertSubject**(const struct DecodedCert * cert, char * buf, word32 * bufSz)Retrieves the subject name from a decoded certificate.
int	**wc_GetDecodedCertIssuer**(const struct DecodedCert * cert, char * buf, word32 * bufSz)Retrieves the issuer name from a decoded certificate.
int	**wc_GetDecodedCertSerial**(const struct DecodedCert * cert, byte * buf, word32 * bufSz)Retrieves the serial number from a decoded certificate.
int	**wc_GetUUIDFromCert**(struct DecodedCert * cert, byte * uuid, int * uuidSz)Extracts UUID from certificate.
int	**wc_GetFASCNFromCert**(struct DecodedCert * cert, byte * fascn, int * fascnSz)Extracts FASCN from certificate.
int	**wc_GeneratePreTBS**(struct DecodedCert * cert, byte * der, int derSz)Generates the pre_TBS (To Be Signed) certificate data from a decoded certificate. The TBS portion is the certificate data that gets signed by the certificate authority. This function is used in dual algorithm certificate creation where the TBS data needs to be extracted for signing with an alternative algorithm (e.g., a post_quantum algorithm).
void	**wc_InitDecodedAcert**(struct DecodedAcert * acert, void * heap)Initializes decoded attribute certificate structure.

	Name
void	**wc_FreeDecodedAcert**(struct DecodedAcert * acert)Frees decoded attribute certificate structure.
int	**wc_ParseX509Acert**(struct DecodedAcert * acert, int verify)Parses X.509 attribute certificate.
int	**wc_VerifyX509Acert**(const byte * acert, word32 acertSz, const byte * issuerCert, word32 issuerCertSz, void * cm)Verifies X.509 attribute certificate.
int	**wc_DecryptPKCS8Key**(byte * input, word32 sz, const char * password, int passwordSz)This function takes an encrypted PKCS#8 DER key and decrypts it to PKCS#8 unencrypted DER.Undoes the encryption done by wc_EncryptPKCS8Key. See RFC5208. The input buffer is overwritten with the decrypted data.
int	**wc_CreateEncryptedPKCS8Key**(byte * key, word32 keySz, byte * out, word32 * outSz, const char * password, int passwordSz, int vPKCS, int pbeOid, int encAlgId, byte * salt, word32 saltSz, int itt, WC_RNG * rng, void * heap)This function takes a traditional, DER key, converts it to PKCS#8 format, and encrypts it. It uses wc_CreatePKCS8Key and wc_EncryptPKCS8Key to do this.
void	**wc_InitDecodedCert**(struct DecodedCert * cert, const byte * source, word32 inSz, void * heap)This function initializes the DecodedCert pointed to by the “cert” parameter. It saves the “source” pointer to a DER-encoded certificate of length “inSz.” This certificate can be parsed by a subsequent call to wc_ParseCert.
int	**wc_ParseCert**(DecodedCert * cert, int type, int verify, WOLFSSL_CERT_MANAGER * cm)This function parses the DER-encoded certificate saved in the DecodedCert object and populates the fields of that object. The DecodedCert must have been initialized with a prior call to wc_InitDecodedCert. This function takes an optional pointer to a CertificateManager object, which is used to populate the certificate authority information of the DecodedCert, if the CA is found in the CertificateManager.
void	**wc_FreeDecodedCert**(struct DecodedCert * cert)This function frees a DecodedCert that was previously initialized with wc_InitDecodedCert.

	Name
int	**wc_SetTimeCb**(wc_time_cb f)This function registers a time callback that will be used anytime wolfSSL needs to get the current time. The prototype of the callback should be the same as the “time” function from the C standard library.
time_t	**wc_Time**(time_t * t)This function gets the current time. By default, it uses the XTIME macro, which varies between platforms. The user can use a function of their choosing instead via the wc_SetTimeCb function.
int	**wc_SetCustomExtension**(Cert * cert, int critical, const char * oid, const byte * der, word32 derSz)This function injects a custom extension in to an X.509 certificate. note: The content at the address pointed to by any of the parameters that are pointers must not be modified until the certificate is generated and you have the der output. This function does NOT copy the contents to another buffer.
int	**wc_SetUnknownExtCallback**(DecodedCert * cert, wc_UnknownExtCallback cb)This function registers a callback that will be used anytime wolfSSL encounters an unknown X.509 extension in a certificate while parsing a certificate. The prototype of the callback should be:
int	**wc_CheckCertSigPubKey**(const byte * cert, word32 certSz, void * heap, const byte * pubKey, word32 pubKeySz, int pubKeyOID)This function verifies the signature in the der form of an X.509 certificate against a public key. The public key is expected to be the full subject public key info in der form.
int	**wc_Asn1PrintOptions_Init**(Asn1PrintOptions * opts)This function initializes the ASN.1 print options.
int	**wc_Asn1PrintOptions_Set**(Asn1PrintOptions * opts, enum Asn1PrintOpt opt, word32 val)This function sets a print option into an ASN.1 print options object.
int	**wc_Asn1_Init**(Asn1 * asn1)This function initializes an ASN.1 parsing object.
int	**wc_Asn1_SetFile**(Asn1 * asn1, XFILE file)This function sets the file to use when printing into an ASN.1 parsing object.
int	**wc_Asn1_PrintAll**(Asn1 * asn1, Asn1PrintOptions * opts, unsigned char * data, word32 len)Print all ASN.1 items.
int	**wc_Asn1_SetOidToNameCb**(Asn1 * asn1, Asn1OidToNameCb nameCb)Sets OID to name callback for ASN.1 parsing.

	Name
int	**wc_GetDecodedCertSubjectRaw**(const struct DecodedCert * cert, const byte ** subjectRaw, int * subjectRawSz)Retrieves the raw DER-encoded subject Name content from a parsed DecodedCert.

C.5.2 Functions Documentation

C.5.2.1 function wc_InitCert

```
int wc_InitCert(
    Cert * cert
)
```

This function initializes a default cert, with the default options: version = 3 (0x2), serial = 0, sigType = SHA_WITH_RSA, issuer = blank, daysValid = 500, selfSigned = 1 (true) use subject as issuer, subject = blank.

Parameters:

- **cert** pointer to an uninitialized cert structure to initialize

See:

- wc_MakeCert
- wc_MakeCertReq

Return: none No returns.

Example

```
Cert myCert;
wc_InitCert(&myCert);
```

C.5.2.2 function wc_CertNew

```
Cert * wc_CertNew(
    void * heap
)
```

This function allocates a new Cert structure for use during cert operations without the application having to allocate the structure itself. The Cert structure is also initialized by this function thus removing the need to call wc_InitCert(). When the application is finished using the allocated Cert structure wc_CertFree() must be called.

Parameters:

- **A** pointer to the heap used for dynamic allocation. Can be NULL.

See:

- wc_InitCert
- wc_MakeCert
- wc_CertFree

Return:

- pointer If successful the call will return a pointer to the newly allocated and initialized Cert.
- NULL On a memory allocation failure.

Example

```
Cert*   myCert;

myCert = wc_CertNew(NULL);
if (myCert == NULL) {
    // Cert creation failure
}
```

C.5.2.3 function wc_InitCert_ex

```
int wc_InitCert_ex(
    Cert * cert,
    void * heap,
    int devId
)
```

Initializes certificate with heap hint and device ID.

Parameters:

- **cert** Cert structure to initialize
- **heap** Heap hint for memory allocation
- **devId** Device ID for hardware acceleration

See:

- wc_InitCert
- wc_MakeCert_ex

Return:

- 0 on success
- BAD_FUNC_ARG if cert is NULL

Example

```
Cert myCert;
int ret = wc_InitCert_ex(&myCert, NULL, INVALID_DEVID);
if (ret != 0) {
    // error initializing cert
}
```

C.5.2.4 function wc_CertFree

```
void wc_CertFree(
    Cert * cert
)
```

This function frees the memory allocated for a cert structure by a previous call to wc_CertNew().

Parameters:

- **A** pointer to the cert structure to free.

See:

- wc_InitCert
- wc_MakeCert
- wc_CertNew

Return: None.

Example

```
Cert*   myCert;

myCert = wc_CertNew(NULL);

// Perform cert operations.

wc_CertFree(myCert);
```

C.5.2.5 function wc_MakeCert

```
int wc_MakeCert(
    Cert * cert,
    byte * derBuffer,
    word32 derSz,
    RsaKey * rsaKey,
    ecc_key * eccKey,
    WC_RNG * rng
)
```

Used to make CA signed certs. Called after the subject information has been entered. This function makes an x509 Certificate v3 RSA or ECC from a cert input. It then writes this cert to derBuffer. It takes in either an rsaKey or an eccKey to generate the certificate. The certificate must be initialized with wc_InitCert before this method is called.

Parameters:

- **cert** pointer to an initialized cert structure
- **derBuffer** pointer to the buffer in which to hold the generated cert
- **derSz** size of the buffer in which to store the cert
- **rsaKey** pointer to an RsaKey structure containing the rsa key used to generate the certificate
- **eccKey** pointer to an EccKey structure containing the ecc key used to generate the certificate
- **rng** pointer to the random number generator used to make the cert

See:

- wc_InitCert
- wc_MakeCertReq

Return:

- Success On successfully making an x509 certificate from the specified input cert, returns the size of the cert generated.
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- BUFFER_E Returned if the provided derBuffer is too small to store the generated certificate
- BAD_FUNC_ARG Returned if cert->serialSz is negative, if the encoded serial would exceed CTC_SERIAL_SIZE octets, or if the serial number is zero. RFC 5280 4.1.2.2 requires a positive serial of at most 20 octets; define WOLFSSL_ASN_ALLOW_0_SERIAL to permit a zero serial.
- Others Additional error messages may be returned if the cert generation is not successful.

A serial number left at the wc_InitCert default (cert->serialSz of 0) is randomly generated. A caller-supplied serial is taken as a big-endian value of at most CTC_SERIAL_SIZE bytes and is normalized to a minimal DER INTEGER: redundant leading zero bytes are stripped and the sign pad is added back when the high bit is set. Supplying the magnitude alone is enough; a sign pad the caller adds is accepted and is not duplicated. The encoded value, including any sign pad, must not exceed CTC_SERIAL_SIZE octets.

Example

```
Cert myCert;
wc_InitCert(&myCert);
WC_RNG rng;
//initialize rng;
RsaKey key;
//initialize key;
byte * derCert = malloc(FOURK_BUF);
word32 certSz;
certSz = wc_MakeCert(&myCert, derCert, FOURK_BUF, &key, NULL, &rng);
```

C.5.2.6 function wc_MakeCert_ex

```
int wc_MakeCert_ex(
    Cert * cert,
    byte * derBuffer,
    word32 derSz,
    int keyType,
    void * key,
    WC_RNG * rng
)
```

Makes certificate with generic key type support.

Parameters:

- **cert** Initialized cert structure
- **derBuffer** Buffer for generated certificate
- **derSz** Size of derBuffer
- **keyType** Key type (RSA_TYPE, ECC_TYPE, ED25519_TYPE, etc.)
- **key** Pointer to key structure
- **rng** Random number generator

See:

- wc_MakeCert
- wc_SignCert_ex

Return:

- Size of certificate on success
- MEMORY_E if memory allocation fails
- BUFFER_E if buffer too small
- BAD_FUNC_ARG if the serial number is zero, negative in size, or longer than CTC_SERIAL_SIZE
- Other error codes on failure

The serial number contract is the same as wc_MakeCert().

Example

```
Cert myCert;
wc_InitCert(&myCert);
byte derCert[4096];
RsaKey key;
WC_RNG rng;
int certSz = wc_MakeCert_ex(&myCert, derCert, sizeof(derCert),
                            RSA_TYPE, &key, &rng);
```

C.5.2.7 function wc_MakeCertReq_ex

```
int wc_MakeCertReq_ex(
    Cert * cert,
    byte * derBuffer,
    word32 derSz,
    int keyType,
    void * key
)
```

Makes certificate request with generic key type support.

Parameters:

- **cert** Initialized cert structure
- **derBuffer** Buffer for generated certificate request
- **derSz** Size of derBuffer
- **keyType** Key type (RSA_TYPE, ECC_TYPE, ED25519_TYPE, etc.)
- **key** Pointer to key structure

See:

- wc_MakeCertReq
- wc_SignCert_ex

Return:

- Size of certificate request on success
- MEMORY_E if memory allocation fails
- BUFFER_E if buffer too small
- Other error codes on failure

Example

```
Cert myCert;
wc_InitCert(&myCert);
byte derCert[4096];
EccKey key;
int certSz = wc_MakeCertReq_ex(&myCert, derCert, sizeof(derCert),
                               ECC_TYPE, &key);
```

C.5.2.8 function wc_SignCert_ex

```
int wc_SignCert_ex(
    int requestSz,
    int sType,
    byte * buf,
    word32 buffSz,
    int keyType,
    void * key,
    WC_RNG * rng
)
```

Signs certificate with generic key type support.

Parameters:

- **requestSz** Size of certificate body to sign
- **sType** Signature type
- **buf** Buffer containing certificate to sign
- **buffSz** Total size of buffer

- **keyType** Key type (RSA_TYPE, ECC_TYPE, ED25519_TYPE, etc.)
- **key** Pointer to key structure
- **rng** Random number generator

See:

- wc_SignCert
- wc_MakeCert_ex

Return:

- New size of certificate with signature on success
- MEMORY_E if memory allocation fails
- BUFFER_E if buffer too small
- Other error codes on failure

Example

```
Cert myCert;
byte derCert[4096];
RsaKey key;
WC_RNG rng;
// Initialize cert and set fields (issuer, subject, dates, etc.)
wc_InitCert(&myCert);
// ... set myCert fields ...
// Generate certificate body (TBS - To Be Signed)
int bodySz = wc_MakeCert_ex(&myCert, derCert, sizeof(derCert),
                            RSA_TYPE, &key, &rng);
if (bodySz > 0) {
    // bodySz is the size of the unsigned certificate body
    // Sign the certificate body and append signature
    int certSz = wc_SignCert_ex(bodySz, CTC_SHA256wRSA,
                                derCert, sizeof(derCert), RSA_TYPE,
                                &key, &rng);
    // derCert now contains complete signed certificate of size certSz
}
```

C.5.2.9 function wc_MakeSigWithBitStr

```
int wc_MakeSigWithBitStr(
    byte * sig,
    int sigSz,
    int sType,
    byte * buf,
    word32 bufSz,
    int keyType,
    void * key,
    WC_RNG * rng
)
```

Makes signature with bit string encoding. This function is used for dual algorithm certificate signing, where an alternative signature is created using a secondary key algorithm (e.g., a post-quantum algorithm alongside a traditional algorithm).

Parameters:

- **sig** Output buffer for signature
- **sigSz** Size of signature buffer
- **sType** Signature type

- **buf** Data to sign (typically the TBS - To Be Signed - certificate data)
- **bufSz** Size of data
- **keyType** Key type (RSA_TYPE, ECC_TYPE, ED25519_TYPE, etc.)
- **key** Pointer to key structure
- **rng** Random number generator

See:

- wc_SignCert_ex
- wc_GeneratePreTBS

Return:

- Size of signature on success
- Negative on error

Note: This API is only available when WOLFSSL_DUAL_ALG_CERTS is defined, which enables support for dual algorithm certificates used in Post-Quantum cryptography to provide hybrid signing with both traditional and PQ algorithms.

Example

```
byte sig[512], data[256];
RsaKey key;
WC_RNG rng;
int sigSz = wc_MakeSigWithBitStr(sig, sizeof(sig), CTC_SHA256wRSA,
                                 data, sizeof(data), RSA_TYPE,
                                 &key, &rng);
```

C.5.2.10 function wc_GetCertDates

```
int wc_GetCertDates(
    Cert * cert,
    struct tm * before,
    struct tm * after
)
```

Gets certificate validity dates.

Parameters:

- **cert** Certificate structure
- **before** Output for notBefore date
- **after** Output for notAfter date

See: wc_InitCert

Return:

- 0 on success
- BAD_FUNC_ARG if parameters invalid

Example

```
Cert myCert;
struct tm beforeDate, afterDate;
int ret = wc_GetCertDates(&myCert, &beforeDate, &afterDate);
```

C.5.2.11 function wc_GetDateInfo

```
int wc_GetDateInfo(
    const byte * certDate,
    int certDateSz,
    const byte ** date,
    byte * format,
    int * length
)
```

Extracts date information from certificate date field. This function parses an ASN.1 encoded date (including tag and length) and returns a pointer to the raw date value bytes, the ASN.1 time type, and the length of the date value.

Parameters:

- **certDate** Certificate date buffer containing ASN.1 encoded date (tag + length + value)
- **certDateSz** Size of certificate date buffer
- **date** Output pointer set to the raw date value bytes (without tag/length)
- **format** Output byte indicating ASN.1 time type: ASN_UTC_TIME (0x17) or ASN_GENERALIZED_TIME (0x18)
- **length** Output length of the raw date value in bytes

See:

- wc_GetCertDates
- wc_GetDateAsCalendarTime

Return:

- 0 on success
- BAD_FUNC_ARG if parameters invalid
- ASN_PARSE_E if date parsing fails

Example

```
const byte* certDate;
const byte* date;
byte format;
int length;
int ret = wc_GetDateInfo(certDate, certDateSz, &date,
                         &format, &length);
if (ret == 0) {
    // date points to raw time bytes, format indicates UTC or
    // Generalized time, length is the number of date value bytes
}
```

C.5.2.12 function wc_GetDateAsCalendarTime

```
int wc_GetDateAsCalendarTime(
    const byte * date,
    int length,
    byte format,
    struct tm * timearg
)
```

Converts certificate date to calendar time structure.

Parameters:

- **date** Date buffer
- **length** Length of date buffer

- **format** Date format (ASN_UTC_TIME or ASN_GENERALIZED_TIME)
- **timearg** Pointer to tm structure to fill

See:

- wc_GetDateInfo
- wc_GetCertDates

Return:

- 0 on success
- BAD_FUNC_ARG if parameters invalid
- ASN_TIME_E if time conversion fails

Example

```
const byte* date;
int length;
byte format;
struct tm timeInfo;
int ret = wc_GetDateAsCalendarTime(date, length, format,
                                   &timeInfo);
```

C.5.2.13 function wc_MakeCertReq

```
int wc_MakeCertReq(
    Cert * cert,
    byte * derBuffer,
    word32 derSz,
    RsaKey * rsaKey,
    ecc_key * eccKey
)
```

This function makes a certificate signing request using the input certificate and writes the output to derBuffer. It takes in either an rsaKey or an eccKey to generate the certificate request. wc_SignCert() will need to be called after this function to sign the certificate request. Please see the wolfCrypt test application (./wolfcrypt/test/test.c) for an example usage of this function.

Parameters:

- **cert** pointer to an initialized cert structure
- **derBuffer** pointer to the buffer in which to hold the generated certificate request
- **derSz** size of the buffer in which to store the certificate request
- **rsaKey** pointer to an RsaKey structure containing the rsa key used to generate the certificate request
- **eccKey** pointer to an EccKey structure containing the ecc key used to generate the certificate request

See:

- wc_InitCert
- wc_MakeCert

Return:

- Success On successfully making an X.509 certificate request from the specified input cert, returns the size of the certificate request generated.
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- BUFFER_E Returned if the provided derBuffer is too small to store the generated certificate
- Other Additional error messages may be returned if the certificate request generation is not successful.

Example

```
Cert myCert;
// initialize myCert
EccKey key;
//initialize key;
byte* derCert = (byte*)malloc(FOURK_BUF);

word32 certSz;
certSz = wc_MakeCertReq(&myCert, derCert, FOURK_BUF, NULL, &key);
```

C.5.2.14 function wc_SignCert

```
int wc_SignCert(
    int requestSz,
    int sigType,
    byte * derBuffer,
    word32 derSz,
    RsaKey * rsaKey,
    ecc_key * eccKey,
    WC_RNG * rng
)
```

This function signs buffer and adds the signature to the end of buffer. It takes in a signature type. Must be called after wc_MakeCert() or wc_MakeCertReq() if creating a CA signed cert.

Parameters:

- **requestSz** the size of the certificate body we're requesting to have signed
- **sType** Type of signature to create. Valid options are: CTC_MD5wRSA, CTC_SHAwRSA, CTC_SHAwECDSA, CTC_SHA256wECDSA, and CTC_SHA256wRSA
- **buffer** pointer to the buffer containing the certificate to be signed. On success: will hold the newly signed certificate
- **buffSz** the (total) size of the buffer in which to store the newly signed certificate
- **rsaKey** pointer to an RsaKey structure containing the rsa key to used to sign the certificate
- **eccKey** pointer to an EccKey structure containing the ecc key to used to sign the certificate
- **rng** pointer to the random number generator used to sign the certificate

See:

- wc_InitCert
- wc_MakeCert

Return:

- Success On successfully signing the certificate, returns the new size of the cert (including signature).
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- BUFFER_E Returned if the provided buffer is too small to store the generated certificate
- Other Additional error messages may be returned if the cert generation is not successful.

Example

```
Cert myCert;
byte* derCert = (byte*)malloc(FOURK_BUF);
// initialize myCert, derCert
RsaKey key;
// initialize key;
WC_RNG rng;
```

```
// initialize rng

word32 certSz;
certSz = wc_SignCert(myCert.bodySz, myCert.sigType,derCert,FOURK_BUF,
&key, NULL,
&rng);
```

C.5.2.15 function wc_MakeSelfCert

```
int wc_MakeSelfCert(
    Cert * cert,
    byte * derBuffer,
    word32 derSz,
    RsaKey * key,
    WC_RNG * rng
)
```

This function is a combination of the previous two functions, wc_MakeCert and wc_SignCert for self signing (the previous functions may be used for CA requests). It makes a certificate, and then signs it, generating a self-signed certificate.

Parameters:

- **cert** pointer to the cert to make and sign
- **buffer** pointer to the buffer in which to hold the signed certificate
- **buffSz** size of the buffer in which to store the signed certificate
- **key** pointer to an RsaKey structure containing the rsa key to used to sign the certificate
- **rng** pointer to the random number generator used to generate and sign the certificate

See:

- wc_InitCert
- wc_MakeCert
- wc_SignCert

Return:

- Success On successfully signing the certificate, returns the new size of the cert.
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- BUFFER_E Returned if the provided buffer is too small to store the generated certificate
- Other Additional error messages may be returned if the cert generation is not successful.

Example

```
Cert myCert;
byte* derCert = (byte*)malloc(FOURK_BUF);
// initialize myCert, derCert
RsaKey key;
// initialize key;
WC_RNG rng;
// initialize rng

word32 certSz;
certSz = wc_MakeSelfCert(&myCert, derCert, FOURK_BUF, &key, NULL, &rng);
```

C.5.2.16 function wc_SetIssuer

```
int wc_SetIssuer(
    Cert * cert,
```

```
    const char * issuerFile
)
```

This function sets the issuer for a certificate to the issuer in the provided pem issuerFile. It also changes the certificate's self-signed attribute to false. The issuer specified in issuerFile is verified prior to setting the cert issuer. This method is used to set fields prior to signing.

Parameters:

- **cert** pointer to the cert for which to set the issuer
- **issuerFile** path of the file containing the pem formatted certificate

See:

- wc_InitCert
- wc_SetSubject
- wc_SetIssuerBuffer

Return:

- 0 Returned on successfully setting the issuer for the certificate
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_PARSE_E Returned if there is an error parsing the cert header file
- ASN_OBJECT_ID_E Returned if there is an error parsing the encryption type from the cert
- ASN_EXPECT_0_E Returned if there is a formatting error in the encryption specification of the cert file
- ASN_BEFORE_DATE_E Returned if the date is before the certificate start date
- ASN_AFTER_DATE_E Returned if the date is after the certificate expiration date
- ASN_BITSTR_E Returned if there is an error parsing a bit string from the certificate
- ECC_CURVE_OID_E Returned if there is an error parsing the ECC key from the certificate
- ASN_UNKNOWN_OID_E Returned if the certificate is using an unknown key object id
- ASN_VERSION_E Returned if the ALLOW_V1_EXTENSIONS option is not defined and the certificate is a V1 or V2 certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension
- ASN_CRIT_EXT_E Returned if an unfamiliar critical extension is encountered in processing the certificate
- ASN_SIG_OID_E Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
- ASN_SIG_CONFIRM_E Returned if confirming the certification signature fails
- ASN_NAME_INVALID_E Returned if the certificate's name is not permitted by the CA name constraints
- ASN_NO_SIGNER_E Returned if there is no CA signer to verify the certificate's authenticity

Example

```
Cert myCert;
// initialize myCert
if(wc_SetIssuer(&myCert, "./path/to/ca-cert.pem") != 0) {
    // error setting issuer
}
```

C.5.2.17 function wc_SetSubject

```
int wc_SetSubject(
    Cert * cert,
    const char * subjectFile
)
```

This function sets the subject for a certificate to the subject in the provided pem subjectFile. This method is used to set fields prior to signing.

Parameters:

- **cert** pointer to the cert for which to set the issuer
- **subjectFile** path of the file containing the pem formatted certificate

See:

- wc_InitCert
- wc_SetIssuer

Return:

- 0 Returned on successfully setting the issuer for the certificate
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_PARSE_E Returned if there is an error parsing the cert header file
- ASN_OBJECT_ID_E Returned if there is an error parsing the encryption type from the cert
- ASN_EXPECT_0_E Returned if there is a formatting error in the encryption specification of the cert file
- ASN_BEFORE_DATE_E Returned if the date is before the certificate start date
- ASN_AFTER_DATE_E Returned if the date is after the certificate expiration date
- ASN_BITSTR_E Returned if there is an error parsing a bit string from the certificate
- ECC_CURVE_OID_E Returned if there is an error parsing the ECC key from the certificate
- ASN_UNKNOWN_OID_E Returned if the certificate is using an unknown key object id
- ASN_VERSION_E Returned if the ALLOW_V1_EXTENSIONS option is not defined and the certificate is a V1 or V2 certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension
- ASN_CRIT_EXT_E Returned if an unfamiliar critical extension is encountered in processing the certificate
- ASN_SIG_OID_E Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
- ASN_SIG_CONFIRM_E Returned if confirming the certification signature fails
- ASN_NAME_INVALID_E Returned if the certificate's name is not permitted by the CA name constraints
- ASN_NO_SIGNER_E Returned if there is no CA signer to verify the certificate's authenticity

Example

```
Cert myCert;
// initialize myCert
if(wc_SetSubject(&myCert, "./path/to/ca-cert.pem") != 0) {
    // error setting subject
}
```

C.5.2.18 function wc_SetSubjectRaw

```
int wc_SetSubjectRaw(
    Cert * cert,
    const byte * der,
    int derSz
)
```

This function sets the raw subject for a certificate from the subject in the provided der buffer. This method is used to set the raw subject field prior to signing.

Parameters:

- **cert** pointer to the cert for which to set the raw subject

- **der** pointer to the buffer containing the der formatted certificate from which to grab the subject
- **derSz** size of the buffer containing the der formatted certificate from which to grab the subject

See:

- wc_InitCert
- wc_SetSubject

Return:

- 0 Returned on successfully setting the subject for the certificate
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_PARSE_E Returned if there is an error parsing the cert header file
- ASN_OBJECT_ID_E Returned if there is an error parsing the encryption type from the cert
- ASN_EXPECT_0_E Returned if there is a formatting error in the encryption specification of the cert file
- ASN_BEFORE_DATE_E Returned if the date is before the certificate start date
- ASN_AFTER_DATE_E Returned if the date is after the certificate expiration date
- ASN_BITSTR_E Returned if there is an error parsing a bit string from the certificate
- ECC_CURVE_OID_E Returned if there is an error parsing the ECC key from the certificate
- ASN_UNKNOWN_OID_E Returned if the certificate is using an unknown key object id
- ASN_VERSION_E Returned if the ALLOW_V1_EXTENSIONS option is not defined and the certificate is a V1 or V2 certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension
- ASN_CRIT_EXT_E Returned if an unfamiliar critical extension is encountered in processing the certificate
- ASN_SIG_OID_E Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
- ASN_SIG_CONFIRM_E Returned if confirming the certification signature fails
- ASN_NAME_INVALID_E Returned if the certificate's name is not permitted by the CA name constraints
- ASN_NO_SIGNER_E Returned if there is no CA signer to verify the certificate's authenticity

Example

```
Cert myCert;
// initialize myCert
byte* der;
der = (byte*)malloc(FOURK_BUF);
// initialize der
if(wc_SetSubjectRaw(&myCert, der, FOURK_BUF) != 0) {
    // error setting subject
}
```

C.5.2.19 function wc_GetSubjectRaw

```
int wc_GetSubjectRaw(
    byte ** subjectRaw,
    Cert * cert
)
```

This function gets the raw subject from the certificate structure.

Parameters:

- **subjectRaw** pointer-pointer to the raw subject upon successful return
- **cert** pointer to the cert from which to get the raw subject

See:

- wc_InitCert
- wc_SetSubjectRaw

Return:

- 0 Returned on successfully getting the subject from the certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension

Example

```
Cert myCert;
byte *subjRaw;
// initialize myCert

if(wc_GetSubjectRaw(&subjRaw, &myCert) != 0) {
    // error setting subject
}
```

C.5.2.20 function wc_SetAltNames

```
int wc_SetAltNames(
    Cert * cert,
    const char * file
)
```

This function sets the alternate names for a certificate to the alternate names in the provided pem file. This is useful in the case that one wishes to secure multiple domains with the same certificate. This method is used to set fields prior to signing.

Parameters:

- **cert** pointer to the cert for which to set the alt names
- **file** path of the file containing the pem formatted certificate

See:

- wc_InitCert
- wc_SetIssuer

Return:

- 0 Returned on successfully setting the alt names for the certificate
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_PARSE_E Returned if there is an error parsing the cert header file
- ASN_OBJECT_ID_E Returned if there is an error parsing the encryption type from the cert
- ASN_EXPECT_0_E Returned if there is a formatting error in the encryption specification of the cert file
- ASN_BEFORE_DATE_E Returned if the date is before the certificate start date
- ASN_AFTER_DATE_E Returned if the date is after the certificate expiration date
- ASN_BITSTR_E Returned if there is an error parsing a bit string from the certificate
- ECC_CURVE_OID_E Returned if there is an error parsing the ECC key from the certificate
- ASN_UNKNOWN_OID_E Returned if the certificate is using an unknown key object id
- ASN_VERSION_E Returned if the ALLOW_V1_EXTENSIONS option is not defined and the certificate is a V1 or V2 certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension
- ASN_CRIT_EXT_E Returned if an unfamiliar critical extension is encountered in processing the certificate
- ASN_SIG_OID_E Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
- ASN_SIG_CONFIRM_E Returned if confirming the certification signature fails

- ASN_NAME_INVALID_E Returned if the certificate's name is not permitted by the CA name constraints
- ASN_NO_SIGNER_E Returned if there is no CA signer to verify the certificate's authenticity

Example

```
Cert myCert;
// initialize myCert
if(wc_SetSubject(&myCert, "./path/to/ca-cert.pem") != 0) {
    // error setting alt names
}
```

C.5.2.21 function wc_SetIssuerBuffer

```
int wc_SetIssuerBuffer(
    Cert * cert,
    const byte * der,
    int derSz
)
```

This function sets the issuer for a certificate from the issuer in the provided der buffer. It also changes the certificate's self-signed attribute to false. This method is used to set fields prior to signing.

Parameters:

- **cert** pointer to the cert for which to set the issuer
- **der** pointer to the buffer containing the der formatted certificate from which to grab the issuer
- **derSz** size of the buffer containing the der formatted certificate from which to grab the issuer

See:

- wc_InitCert
- wc_SetIssuer

Return:

- 0 Returned on successfully setting the issuer for the certificate
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_PARSE_E Returned if there is an error parsing the cert header file
- ASN_OBJECT_ID_E Returned if there is an error parsing the encryption type from the cert
- ASN_EXPECT_0_E Returned if there is a formatting error in the encryption specification of the cert file
- ASN_BEFORE_DATE_E Returned if the date is before the certificate start date
- ASN_AFTER_DATE_E Returned if the date is after the certificate expiration date
- ASN_BITSTR_E Returned if there is an error parsing a bit string from the certificate
- ECC_CURVE_OID_E Returned if there is an error parsing the ECC key from the certificate
- ASN_UNKNOWN_OID_E Returned if the certificate is using an unknown key object id
- ASN_VERSION_E Returned if the ALLOW_V1_EXTENSIONS option is not defined and the certificate is a V1 or V2 certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension
- ASN_CRIT_EXT_E Returned if an unfamiliar critical extension is encountered in processing the certificate
- ASN_SIG_OID_E Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
- ASN_SIG_CONFIRM_E Returned if confirming the certification signature fails
- ASN_NAME_INVALID_E Returned if the certificate's name is not permitted by the CA name constraints
- ASN_NO_SIGNER_E Returned if there is no CA signer to verify the certificate's authenticity

Example

```
Cert myCert;
// initialize myCert
byte* der;
der = (byte*)malloc(FOURK_BUF);
// initialize der
if(wc_SetIssuerBuffer(&myCert, der, FOURK_BUF) != 0) {
    // error setting issuer
}
```

C.5.2.22 function wc_SetIssuerRaw

```
int wc_SetIssuerRaw(
    Cert * cert,
    const byte * der,
    int derSz
)
```

This function sets the raw issuer for a certificate from the issuer in the provided der buffer. This method is used to set the raw issuer field prior to signing.

Parameters:

- **cert** pointer to the cert for which to set the raw issuer
- **der** pointer to the buffer containing the der formatted certificate from which to grab the subject
- **derSz** size of the buffer containing the der formatted certificate from which to grab the subject

See:

- wc_InitCert
- wc_SetIssuer

Return:

- 0 Returned on successfully setting the issuer for the certificate
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_PARSE_E Returned if there is an error parsing the cert header file
- ASN_OBJECT_ID_E Returned if there is an error parsing the encryption type from the cert
- ASN_EXPECT_0_E Returned if there is a formatting error in the encryption specification of the cert file
- ASN_BEFORE_DATE_E Returned if the date is before the certificate start date
- ASN_AFTER_DATE_E Returned if the date is after the certificate expiration date
- ASN_BITSTR_E Returned if there is an error parsing a bit string from the certificate
- ECC_CURVE_OID_E Returned if there is an error parsing the ECC key from the certificate
- ASN_UNKNOWN_OID_E Returned if the certificate is using an unknown key object id
- ASN_VERSION_E Returned if the ALLOW_V1_EXTENSIONS option is not defined and the certificate is a V1 or V2 certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension
- ASN_CRIT_EXT_E Returned if an unfamiliar critical extension is encountered in processing the certificate
- ASN_SIG_OID_E Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
- ASN_SIG_CONFIRM_E Returned if confirming the certification signature fails
- ASN_NAME_INVALID_E Returned if the certificate's name is not permitted by the CA name constraints
- ASN_NO_SIGNER_E Returned if there is no CA signer to verify the certificate's authenticity

Example

```
Cert myCert;
// initialize myCert
byte* der;
der = (byte*)malloc(FOURK_BUF);
// initialize der
if(wc_SetIssuerRaw(&myCert, der, FOURK_BUF) != 0) {
    // error setting subject
}
```

C.5.2.23 function wc_SetSubjectBuffer

```
int wc_SetSubjectBuffer(
    Cert * cert,
    const byte * der,
    int derSz
)
```

This function sets the subject for a certificate from the subject in the provided der buffer. This method is used to set fields prior to signing.

Parameters:

- **cert** pointer to the cert for which to set the subject
- **der** pointer to the buffer containing the der formatted certificate from which to grab the subject
- **derSz** size of the buffer containing the der formatted certificate from which to grab the subject

See:

- wc_InitCert
- wc_SetSubject

Return:

- 0 Returned on successfully setting the subject for the certificate
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_PARSE_E Returned if there is an error parsing the cert header file
- ASN_OBJECT_ID_E Returned if there is an error parsing the encryption type from the cert
- ASN_EXPECT_0_E Returned if there is a formatting error in the encryption specification of the cert file
- ASN_BEFORE_DATE_E Returned if the date is before the certificate start date
- ASN_AFTER_DATE_E Returned if the date is after the certificate expiration date
- ASN_BITSTR_E Returned if there is an error parsing a bit string from the certificate
- ECC_CURVE_OID_E Returned if there is an error parsing the ECC key from the certificate
- ASN_UNKNOWN_OID_E Returned if the certificate is using an unknown key object id
- ASN_VERSION_E Returned if the ALLOW_V1_EXTENSIONS option is not defined and the certificate is a V1 or V2 certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension
- ASN_CRIT_EXT_E Returned if an unfamiliar critical extension is encountered in processing the certificate
- ASN_SIG_OID_E Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
- ASN_SIG_CONFIRM_E Returned if confirming the certification signature fails
- ASN_NAME_INVALID_E Returned if the certificate's name is not permitted by the CA name constraints
- ASN_NO_SIGNER_E Returned if there is no CA signer to verify the certificate's authenticity

Example

```
Cert myCert;
// initialize myCert
byte* der;
der = (byte*)malloc(FOURK_BUF);
// initialize der
if(wc_SetSubjectBuffer(&myCert, der, FOURK_BUF) != 0) {
    // error setting subject
}
```

C.5.2.24 function wc_SetAltNamesBuffer

```
int wc_SetAltNamesBuffer(
    Cert * cert,
    const byte * der,
    int derSz
)
```

This function sets the alternate names for a certificate from the alternate names in the provided der buffer. This is useful in the case that one wishes to secure multiple domains with the same certificate. This method is used to set fields prior to signing.

Parameters:

- **cert** pointer to the cert for which to set the alternate names
- **der** pointer to the buffer containing the der formatted certificate from which to grab the alternate names
- **derSz** size of the buffer containing the der formatted certificate from which to grab the alternate names

See:

- wc_InitCert
- wc_SetAltNames

Return:

- 0 Returned on successfully setting the alternate names for the certificate
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_PARSE_E Returned if there is an error parsing the cert header file
- ASN_OBJECT_ID_E Returned if there is an error parsing the encryption type from the cert
- ASN_EXPECT_0_E Returned if there is a formatting error in the encryption specification of the cert file
- ASN_BEFORE_DATE_E Returned if the date is before the certificate start date
- ASN_AFTER_DATE_E Returned if the date is after the certificate expiration date
- ASN_BITSTR_E Returned if there is an error parsing a bit string from the certificate
- ECC_CURVE_OID_E Returned if there is an error parsing the ECC key from the certificate
- ASN_UNKNOWN_OID_E Returned if the certificate is using an unknown key object id
- ASN_VERSION_E Returned if the ALLOW_V1_EXTENSIONS option is not defined and the certificate is a V1 or V2 certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension
- ASN_CRIT_EXT_E Returned if an unfamiliar critical extension is encountered in processing the certificate
- ASN_SIG_OID_E Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
- ASN_SIG_CONFIRM_E Returned if confirming the certification signature fails
- ASN_NAME_INVALID_E Returned if the certificate's name is not permitted by the CA name constraints
- ASN_NO_SIGNER_E Returned if there is no CA signer to verify the certificate's authenticity

Example

```
Cert myCert;
// initialize myCert
byte* der;
der = (byte*)malloc(FOURK_BUF);
// initialize der
if(wc_SetAltNamesBuffer(&myCert, der, FOURK_BUF) != 0) {
    // error setting subject
}
```

C.5.2.25 function wc_SetDNSEntry

```
int wc_SetDNSEntry(
    void * heap,
    const char * str,
    int strLen,
    int type,
    DNS_entry ** entries
)
```

This function allocates a single subject alternative name (SAN) entry, copies the supplied name into it, sets its GeneralName type and length, and appends it to the linked list pointed to by entries. The name is duplicated internally, so the caller's buffer need not outlive the call. The resulting list can be encoded with wc_FlattenAltNames and must be freed with FreeAltNames.

Parameters:

- **heap** pointer to the heap hint used for allocations (may be NULL)
- **str** pointer to the name bytes (e.g. a DNS string, or raw IP octets for ASN_IP_TYPE)
- **strLen** length of str in bytes
- **type** GeneralName type (e.g. ASN_DNS_TYPE, ASN_IP_TYPE, ASN_RFC822_TYPE, ASN_URI_TYPE)
- **entries** in/out pointer to the head of the alt-name linked list; a new entry is appended

See:

- wc_FlattenAltNames
- FreeAltNames

Return:

- 0 Returned on success.
- MEMORY_E Returned if dynamic memory allocation fails.
- BAD_FUNC_ARG Returned if str or entries is NULL, strLen is negative, or type is not a supported GeneralName type; also returned for an ASN_IP_TYPE entry whose length is not a valid IPv4/IPv6 address, or an ASN_RID_TYPE entry with malformed contents.
- BUFFER_E Returned if the string representation of an ASN_IP_TYPE or ASN_RID_TYPE entry does not fit its internal buffer.
- Other negative error codes may propagate from generating the string form of ASN_IP_TYPE and ASN_RID_TYPE entries.

Note:

- This helper (along with wc_FlattenAltNames and FreeAltNames) is exported from the library only when WOLFSSL_PUBLIC_ASN, OPENSSL_EXTRA, OPENSSL_EXTRA_X509_SMALL, or WOLFSSL_TEST_CERT is defined; its prototype lives in wolfssl/wolfcrypt/asn.h (not asn_public.h) because it uses the DNS_entry type.

- This function additionally requires WOLFSSL_ASN_TEMPLATE (its internal SetDNSEntry/AddDNSEntryToList helpers are template-only), on top of the WOLFSSL_CERT_GEN && WOLFSSL_ALT_NAMES gating shared with its companions wc_FlattenAltNames and wc_SetAltNamesFromList, which do not require WOLFSSL_ASN_TEMPLATE. Because this builder is the more restrictive of the set, a DNS_entry list built here can always be encoded by a public API in the same build; in non-template builds the public list can only be sourced from a parsed DecodedCert.altNames.

Example

```
DNS_entry* list = NULL;
if (wc_SetDNSEntry(NULL, "example.com", 11, ASN_DNS_TYPE, &list) != 0) {
    // error adding alt name
}
// ... encode with wc_FlattenAltNames, then:
FreeAltNames(list, NULL);
```

C.5.2.26 function wc_FlattenAltNames

```
int wc_FlattenAltNames(
    byte * output,
    word32 outputSz,
    const DNS_entry * names
)
```

This function encodes a linked list of subject alternative name entries into the DER GeneralNames SEQUENCE used as the value of the subjectAltName certificate extension. The output is suitable for assigning to Cert.altNames (with the return value stored in Cert.altNamesSz) prior to signing.

Parameters:

- **output** buffer that receives the DER GeneralNames SEQUENCE; size it to hold the full extension value (e.g. CTC_MAX_ALT_SIZE)
- **outputSz** capacity of output in bytes
- **names** head of the alt-name linked list to encode (e.g. built with wc_SetDNSEntry, or taken from a parsed DecodedCert.altNames)

See:

- wc_SetDNSEntry
- wc_SetAltNamesFromList
- FreeAltNames
- wc_SetAltNamesBuffer

Return:

- 0 the number of bytes written to output (the full SEQUENCE, including its tag and length).
- 0 Returned when names is NULL (nothing to encode).
- BAD_FUNC_ARG Returned if output is NULL.
- BUFFER_E Returned if output is too small to hold the encoding.

Note: This helper (along with wc_SetDNSEntry and FreeAltNames) is exported from the library only when WOLFSSL_PUBLIC_ASN, OPENSSL_EXTRA, OPENSSL_EXTRA_X509_SMALL, or WOLFSSL_TEST_CERT is defined; its prototype lives in wolfssl/wolfcrypt/asn.h (not asn_public.h) because it uses the DNS_entry type.

Example

```
Cert cert;
DNS_entry* list = NULL;
// ... populate list with wc_SetDNSEntry ...
int n = wc_FlattenAltNames(cert.altNames, sizeof(cert.altNames), list);
if (n < 0) {
    // error encoding alt names
}
cert.altNamesSz = n;
FreeAltNames(list, NULL);
```

C.5.2.27 function wc_SetAltNamesFromList

```
int wc_SetAltNamesFromList(
    Cert * cert,
    const DNS_entry * names
)
```

This function encodes a linked list of subject alternative name entries directly into a Cert structure, ready for signing. It is a convenience wrapper around wc_FlattenAltNames: the list is encoded into cert->altNames and the encoded length is stored in cert->altNamesSz, so the caller does not have to manage the buffer or size bookkeeping. The supplied list is not consumed and must still be freed by the caller with FreeAltNames.

Parameters:

- **cert** pointer to the Cert whose altNames/altNamesSz fields are set
- **names** head of the alt-name linked list to encode (e.g. built with wc_SetDNSEntry, or taken from a parsed DecodedCert.altNames); may be NULL, in which case cert->altNamesSz is set to 0

See:

- wc_SetDNSEntry
- wc_FlattenAltNames
- wc_SetAltNamesBuffer
- FreeAltNames

Return:

- 0 Returned on success.
- BAD_FUNC_ARG Returned if cert is NULL.
- BUFFER_E Returned if the encoded names do not fit in cert->altNames.

Note: This helper (along with wc_SetDNSEntry, wc_FlattenAltNames, and FreeAltNames) is exported from the library only when WOLFSSL_PUBLIC_ASN, OPENSSL_EXTRA, OPENSSL_EXTRA_X509_SMALL, or WOLFSSL_TEST_CERT is defined; its prototype lives in wolfssl/wolfcrypt/asn.h (not asn_public.h) because it uses the DNS_entry type. Its DER-input sibling wc_SetAltNamesBuffer is always exported.

Example

```
Cert cert;
DNS_entry* list = NULL;
wc_InitCert(&cert);
// ... populate list with wc_SetDNSEntry ...
if (wc_SetAltNamesFromList(&cert, list) != 0) {
    // error encoding alt names
}
FreeAltNames(list, NULL);
// ... wc_MakeCert / wc_SignCert ...
```

C.5.2.28 function wc_SetDatesBuffer

```
int wc_SetDatesBuffer(
    Cert * cert,
    const byte * der,
    int derSz
)
```

This function sets the dates for a certificate from the date range in the provided der buffer. This method is used to set fields prior to signing.

Parameters:

- **cert** pointer to the cert for which to set the dates
- **der** pointer to the buffer containing the der formatted certificate from which to grab the date range
- **derSz** size of the buffer containing the der formatted certificate from which to grab the date range

See: wc_InitCert

Return:

- 0 Returned on successfully setting the dates for the certificate
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_PARSE_E Returned if there is an error parsing the cert header file
- ASN_OBJECT_ID_E Returned if there is an error parsing the encryption type from the cert
- ASN_EXPECT_0_E Returned if there is a formatting error in the encryption specification of the cert file
- ASN_BEFORE_DATE_E Returned if the date is before the certificate start date
- ASN_AFTER_DATE_E Returned if the date is after the certificate expiration date
- ASN_BITSTR_E Returned if there is an error parsing a bit string from the certificate
- ECC_CURVE_OID_E Returned if there is an error parsing the ECC key from the certificate
- ASN_UNKNOWN_OID_E Returned if the certificate is using an unknown key object id
- ASN_VERSION_E Returned if the ALLOW_V1_EXTENSIONS option is not defined and the certificate is a V1 or V2 certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension
- ASN_CRIT_EXT_E Returned if an unfamiliar critical extension is encountered in processing the certificate
- ASN_SIG_OID_E Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
- ASN_SIG_CONFIRM_E Returned if confirming the certification signature fails
- ASN_NAME_INVALID_E Returned if the certificate's name is not permitted by the CA name constraints
- ASN_NO_SIGNER_E Returned if there is no CA signer to verify the certificate's authenticity

Example

```
Cert myCert;
// initialize myCert
byte* der;
der = (byte*)malloc(FOURK_BUF);
// initialize der
if(wc_SetDatesBuffer(&myCert, der, FOURK_BUF) != 0) {
    // error setting subject
}
```

C.5.2.29 function wc_SetAuthKeyIdFromPublicKey

```
int wc_SetAuthKeyIdFromPublicKey(
    Cert * cert,
    RsaKey * rsakey,
    ecc_key * eckey
)
```

Set AKID from either an RSA or ECC public key. note: Only set one of rsakey or eckey, not both.

Parameters:

- **cert** Pointer to the certificate to set the SKID.
- **rsakey** Pointer to the RsaKey struct to read from.
- **eckey** Pointer to the ecc_key to read from.

See:

- wc_SetSubjectKeyId
- wc_SetAuthKeyId
- wc_SetAuthKeyIdFromCert

Return:

- 0 Success
- BAD_FUNC_ARG Either cert is null or both rsakey and eckey are null.
- MEMORY_E Error allocating memory.
- PUBLIC_KEY_E Error writing to the key.

Example

```
Cert myCert;
RsaKey keypub;

wc_InitRsaKey(&keypub, 0);

if (wc_SetAuthKeyIdFromPublicKey(&myCert, &keypub, NULL) != 0)
{
    // Handle error
}
```

C.5.2.30 function wc_SetAuthKeyIdFromPublicKey_ex

```
int wc_SetAuthKeyIdFromPublicKey_ex(
    Cert * cert,
    int keyType,
    void * key
)
```

Sets authority key ID from public key with generic key type.

Parameters:

- **cert** Certificate structure
- **keyType** Key type (RSA_TYPE, ECC_TYPE, ED25519_TYPE, etc.)
- **key** Pointer to key structure

See: wc_SetAuthKeyIdFromPublicKey

Return:

- 0 on success
- BAD_FUNC_ARG if parameters invalid
- MEMORY_E if memory allocation fails

Example

```
Cert myCert;
wc_InitCert(&myCert);
RsaKey key;
int ret = wc_SetAuthKeyIdFromPublicKey_ex(&myCert, RSA_TYPE,
                                          &key);
```

C.5.2.31 function wc_SetAuthKeyIdFromCert

```
int wc_SetAuthKeyIdFromCert(
    Cert * cert,
    const byte * der,
    int derSz
)
```

Set AKID from from DER encoded certificate.

Parameters:

- **cert** The Cert struct to write to.
- **der** The DER encoded certificate buffer.
- **derSz** Size of der in bytes.

See:

- wc_SetAuthKeyIdFromPublicKey
- wc_SetAuthKeyId

Return:

- 0 Success
- BAD_FUNC_ARG Error if any argument is null or derSz is less than 0.
- MEMORY_E Error if problem allocating memory.
- ASN_NO_SKID No subject key ID found.

Example

```
Cert some_cert;
byte some_der[] = { // Initialize a DER buffer };
wc_InitCert(&some_cert);
if(wc_SetAuthKeyIdFromCert(&some_cert, some_der, sizeof(some_der) != 0)
{
    // Handle error
}
```

C.5.2.32 function wc_SetAuthKeyId

```
int wc_SetAuthKeyId(
    Cert * cert,
    const char * file
)
```

Set AKID from certificate file in PEM format.

Parameters:

- **cert** Cert struct you want to set the AKID of.
- **file** Buffer containing PEM cert file.

See:

- wc_SetAuthKeyIdFromPublicKey
- wc_SetAuthKeyIdFromCert

Return:

- 0 Success
- BAD_FUNC_ARG Error if cert or file is null.
- MEMORY_E Error if problem allocating memory.

Example

```
char* file_name = "/path/to/file";
cert some_cert;
wc_InitCert(&some_cert);

if(wc_SetAuthKeyId(&some_cert, file_name) != 0)
{
    // Handle Error
}
```

C.5.2.33 function wc_SetSubjectKeyIdFromPublicKey

```
int wc_SetSubjectKeyIdFromPublicKey(
    Cert * cert,
    RsaKey * rsakey,
    ecc_key * eckey
)
```

Set SKID from RSA or ECC public key.

Parameters:

- **cert** Pointer to a Cert structure to be used.
- **rsakey** Pointer to an RsaKey structure
- **eckey** Pointer to an ecc_key structure

See: wc_SetSubjectKeyId

Return:

- 0 Success
- BAD_FUNC_ARG Returned if cert or rsakey and eckey are null.
- MEMORY_E Returned if there is an error allocating memory.
- PUBLIC_KEY_E Returned if there is an error getting the public key.

Example

```
Cert some_cert;
RsaKey some_key;
wc_InitCert(&some_cert);
wc_InitRsaKey(&some_key);

if(wc_SetSubjectKeyIdFromPublicKey(&some_cert,&some_key, NULL) != 0)
{
    // Handle Error
}
```

C.5.2.34 function wc_SetSubjectKeyIdFromPublicKey_ex

```
int wc_SetSubjectKeyIdFromPublicKey_ex(
    Cert * cert,
    int keyType,
    void * key
)
```

Sets subject key ID from public key with generic key type.

Parameters:

- **cert** Certificate structure
- **keyType** Key type (RSA_TYPE, ECC_TYPE, ED25519_TYPE, etc.)
- **key** Pointer to key structure

See: wc_SetSubjectKeyIdFromPublicKey

Return:

- 0 on success
- BAD_FUNC_ARG if parameters invalid
- MEMORY_E if memory allocation fails
- PUBLIC_KEY_E if error getting public key

Example

```
Cert myCert;
wc_InitCert(&myCert);
EccKey key;
int ret = wc_SetSubjectKeyIdFromPublicKey_ex(&myCert, ECC_TYPE,
                                             &key);
```

C.5.2.35 function wc_SetSubjectKeyId

```
int wc_SetSubjectKeyId(
    Cert * cert,
    const char * file
)
```

Set SKID from public key file in PEM format. Both arguments are required.

Parameters:

- **cert** Cert structure to set the SKID of.
- **file** Contains the PEM encoded file.

See: wc_SetSubjectKeyIdFromPublicKey

Return:

- 0 Success
- BAD_FUNC_ARG Returns if cert or file is null.
- MEMORY_E Returns if there is a problem allocating memory for key.
- PUBLIC_KEY_E Returns if there is an error decoding the public key.

Example

```
const char* file_name = "path/to/file";
Cert some_cert;
wc_InitCert(&some_cert);

if(wc_SetSubjectKeyId(&some_cert, file_name) != 0)
{
```

```
    // Handle Error
}
```

C.5.2.36 function wc_SetKeyUsage

```
int wc_SetKeyUsage(
    Cert * cert,
    const char * value
)
```

This function allows you to set the key usage using a comma delimited string of tokens. Accepted tokens are: digitalSignature, nonRepudiation, contentCommitment, keyCertSign, cRLSign, dataEncipherment, keyAgreement, keyEncipherment, encipherOnly, decipherOnly. Example: "digitalSignature,nonRepudiation" nonRepudiation and contentCommitment are for the same usage.

Parameters:

- **cert** Pointer to initialized Cert structure.
- **value** Comma delimited string of tokens to set usage.

See:

- wc_InitCert
- wc_MakeRsaKey

Return:

- 0 Success
- BAD_FUNC_ARG Returned when either arg is null.
- MEMORY_E Returned when there is an error allocating memory.
- KEYUSAGE_E Returned if an unrecognized token is entered.

Example

```
Cert cert;
wc_InitCert(&cert);

if(wc_SetKeyUsage(&cert, "cRLSign,keyCertSign") != 0)
{
    // Handle error
}
```

C.5.2.37 function wc_SetExtKeyUsage

```
int wc_SetExtKeyUsage(
    Cert * cert,
    const char * value
)
```

Sets extended key usage using comma-delimited string.

Parameters:

- **cert** Certificate structure
- **value** Comma-delimited string of extended key usage values

See:

- wc_SetKeyUsage
- wc_SetExtKeyUsageOID

Return:

- 0 on success
- BAD_FUNC_ARG if parameters invalid
- MEMORY_E if memory allocation fails

Example

```
Cert myCert;
wc_InitCert(&myCert);
int ret = wc_SetExtKeyUsage(&myCert,
                            "serverAuth,clientAuth");
```

C.5.2.38 function wc_SetExtKeyUsageOID

```
int wc_SetExtKeyUsageOID(
    Cert * cert,
    const char * oid,
    word32 sz,
    byte idx,
    void * heap
)
```

Sets extended key usage using OID string.

Parameters:

- **cert** Certificate structure
- **oid** OID string
- **sz** Length of OID string
- **idx** Index for multiple OIDs
- **heap** Heap hint for memory allocation

See: wc_SetExtKeyUsage

Return:

- 0 on success
- BAD_FUNC_ARG if parameters invalid
- MEMORY_E if memory allocation fails

Example

```
Cert myCert;
wc_InitCert(&myCert);
const char* oid = "1.3.6.1.5.5.7.3.1";
int ret = wc_SetExtKeyUsageOID(&myCert, oid, strlen(oid),
                               0, NULL);
```

C.5.2.39 function wc_PemPubKeyToDer

```
int wc_PemPubKeyToDer(
    const char * fileName,
    unsigned char * derBuf,
    int derSz
)
```

Loads a PEM key from a file and converts to a DER encoded buffer.

Parameters:

- **fileName** Name of the file to load.
- **derBuf** Buffer for DER encoded key.

- **derSz** Size of DER buffer.

See: wc_PubKeyPemToDer

Return:

- 0 Success
- <0 Error
- SSL_BAD_FILE There is a problem with opening the file.
- MEMORY_E There is an error allocating memory for the file buffer.
- BUFFER_E derBuf is not large enough to hold the converted key.

Example

```
char* some_file = "filename";
unsigned char der[];

if(wc_PemPubKeyToDer(some_file, der, sizeof(der)) != 0)
{
    //Handle Error
}
```

C.5.2.40 function wc_PemPubKeyToDer_ex

```
int wc_PemPubKeyToDer_ex(
    const char * fileName,
    DerBuffer ** der
)
```

Loads PEM public key from file to DER buffer.

Parameters:

- **fileName** Path to PEM file
- **der** Pointer to DerBuffer pointer to allocate

See:

- wc_PemPubKeyToDer
- wc_FreeDer

Return:

- 0 on success
- negative on error

Example

```
DerBuffer* der = NULL;
int ret = wc_PemPubKeyToDer_ex("pubkey.pem", &der);
if (ret == 0) {
    // Use der->buffer and der->length
    wc_FreeDer(&der);
}
```

C.5.2.41 function wc_PubKeyPemToDer

```
int wc_PubKeyPemToDer(
    const unsigned char * pem,
    int pemSz,
    unsigned char * buff,
```

```
    int buffSz
)
```

Convert a PEM encoded public key to DER. Returns the number of bytes written to the buffer or a negative value for an error.

Parameters:

- **pem** PEM encoded key
- **pemSz** Size of pem
- **buff** Pointer to buffer for output.
- **buffSz** Size of buffer.

See: wc_PemPubKeyToDer

Return:

- 0 Success, number of bytes written.
- BAD_FUNC_ARG Returns if pem, buff, or buffSz are null
- <0 An error occurred in the function.

Example

```
byte some_pem[] = { Initialize with PEM key }
unsigned char out_buffer[1024]; // Ensure buffer is large enough to fit DER

if(wc_PubKeyPemToDer(some_pem, sizeof(some_pem), out_buffer,
sizeof(out_buffer)) < 0)
{
    // Handle error
}
```

C.5.2.42 function wc_PemGetHeaderFooter

```
int wc_PemGetHeaderFooter(
    int type,
    const char ** header,
    const char ** footer
)
```

Gets PEM header and footer strings for given type.

Parameters:

- **type** PEM type (CERT_TYPE, PRIVATEKEY_TYPE, etc.)
- **header** Pointer to header string pointer
- **footer** Pointer to footer string pointer

See: wc_PemToDer

Return:

- 0 on success
- BAD_FUNC_ARG if parameters invalid

Example

```
const char* header;
const char* footer;
int ret = wc_PemGetHeaderFooter(CERT_TYPE, &header, &footer);
```

C.5.2.43 function wc_AllocDer

```
int wc_AllocDer(
    DerBuffer ** pDer,
    word32 length,
    int type,
    void * heap
)
```

Allocates DER buffer with specified length and type.

Parameters:

- **pDer** Pointer to DerBuffer pointer to allocate
- **length** Length of buffer to allocate
- **type** Buffer type for tracking
- **heap** Heap hint for memory allocation

See: wc_FreeDer

Return:

- 0 on success
- BAD_FUNC_ARG if pDer is NULL
- MEMORY_E if allocation fails

Example

```
DerBuffer* der = NULL;
int ret = wc_AllocDer(&der, 1024, CERT_TYPE, NULL);
if (ret == 0) {
    // Use der->buffer
    wc_FreeDer(&der);
}
```

C.5.2.44 function wc_FreeDer

```
void wc_FreeDer(
    DerBuffer ** pDer
)
```

Frees DER buffer allocated by wc_AllocDer or wc_PemToDer.

Parameters:

- **pDer** Pointer to DerBuffer pointer to free

See:

- wc_AllocDer
- wc_PemToDer

Example

```
DerBuffer* der = NULL;
wc_AllocDer(&der, 1024, CERT_TYPE, NULL);
// Use der
wc_FreeDer(&der);
```

C.5.2.45 function wc_PemToDer

```
int wc_PemToDer(
    const unsigned char * buff,
    long longSz,
    int type,
    DerBuffer ** pDer,
    void * heap,
    EncryptedInfo * info,
    int * keyFormat
)
```

Converts PEM to DER format with encryption info support.

Parameters:

- **buff** PEM buffer
- **longSz** Size of PEM buffer
- **type** PEM type (CERT_TYPE, PRIVATEKEY_TYPE, etc.)
- **pDer** Pointer to DerBuffer pointer to allocate
- **heap** Heap hint for memory allocation
- **info** Encryption info for encrypted PEM
- **keyFormat** Pointer to store key format

See:

- wc_PemCertToDer
- wc_FreeDer

Return:

- 0 on success
- negative on error

Example

```
const unsigned char* pem;
DerBuffer* der = NULL;
EncryptedInfo info;
int keyFormat;
int ret = wc_PemToDer(pem, pemSz, PRIVATEKEY_TYPE, &der,
                      NULL, &info, &keyFormat);
if (ret == 0) {
    wc_FreeDer(&der);
}
```

C.5.2.46 function wc_PemCertToDer

```
int wc_PemCertToDer(
    const char * fileName,
    unsigned char * derBuf,
    int derSz
)
```

This function converts a pem certificate to a der certificate, and places the resulting certificate in the derBuf buffer provided.

Parameters:

- **fileName** path to the file containing a pem certificate to convert to a der certificate
- **derBuf** pointer to a char buffer in which to store the converted certificate

- **derSz** size of the char buffer in which to store the converted certificate

See: none

Return:

- Success On success returns the size of the derBuf generated
- BUFFER_E Returned if the size of derBuf is too small to hold the certificate generated
- MEMORY_E Returned if the call to XMALLOC fails

Example

```
char * file = "./certs/client-cert.pem";
int derSz;
byte* der = (byte*)XMALLOC((8*1024), NULL, DYNAMIC_TYPE_CERT);

derSz = wc_PemCertToDer(file, der, (8*1024));
if (derSz <= 0) {
    //PemCertToDer error
}
```

C.5.2.47 function wc_DerToPem

```
int wc_DerToPem(
    const byte * der,
    word32 derSz,
    byte * output,
    word32 outSz,
    int type
)
```

This function converts a der formatted input certificate, contained in the der buffer, into a pem formatted output certificate, contained in the output buffer. It should be noted that this is not an in place conversion, and a separate buffer must be utilized to store the pem formatted output.

Parameters:

- **der** pointer to the buffer of the certificate to convert
- **derSz** size of the the certificate to convert
- **output** pointer to the buffer in which to store the pem formatted certificate
- **outSz** size of the buffer in which to store the pem formatted certificate
- **type** the type of certificate to generate. Valid types are: CERT_TYPE, PRIVATEKEY_TYPE, ECC_PRIVATEKEY_TYPE, and CERTREQ_TYPE.

See: wc_PemCertToDer

Return:

- Success On successfully making a pem certificate from the input der cert, returns the size of the pem cert generated.
- BAD_FUNC_ARG Returned if there is an error parsing the der file and storing it as a pem file
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_INPUT_E Returned in the case of a base64 encoding error
- BUFFER_E May be returned if the output buffer is too small to store the pem formatted certificate

Example

```
byte* der;
// initialize der with certificate
byte* pemFormatted[FOURK_BUF];
```

```
word32 pemSz;
pemSz = wc_DerToPem(der, derSz,pemFormatted,FOURK_BUF, CERT_TYPE);
```

C.5.2.48 function wc_DerToPemEx

```
int wc_DerToPemEx(
    const byte * der,
    word32 derSz,
    byte * output,
    word32 outSz,
    byte * cipher_info,
    int type
)
```

This function converts a der formatted input certificate, contained in the der buffer, into a pem formatted output certificate, contained in the output buffer. It should be noted that this is not an in place conversion, and a separate buffer must be utilized to store the pem formatted output. Allows setting cipher info.

Parameters:

- **der** pointer to the buffer of the certificate to convert
- **derSz** size of the the certificate to convert
- **output** pointer to the buffer in which to store the pem formatted certificate
- **outSz** size of the buffer in which to store the pem formatted certificate
- **cipher_info** Additional cipher information.
- **type** the type of certificate to generate. Valid types are: CERT_TYPE, PRIVATEKEY_TYPE, ECC_PRIVATEKEY_TYPE, and CERTREQ_TYPE.

See: wc_PemCertToDer

Return:

- Success On successfully making a pem certificate from the input der cert, returns the size of the pem cert generated.
- BAD_FUNC_ARG Returned if there is an error parsing the der file and storing it as a pem file
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_INPUT_E Returned in the case of a base64 encoding error
- BUFFER_E May be returned if the output buffer is too small to store the pem formatted certificate

Example

```
byte* der;
// initialize der with certificate
byte* pemFormatted[FOURK_BUF];

word32 pemSz;
byte* cipher_info[] { Additional cipher info. }
pemSz = wc_DerToPemEx(der, derSz, pemFormatted, FOURK_BUF, cipher_info,
 ↪  CERT_TYPE);
```

C.5.2.49 function wc_KeyPemToDer

```
int wc_KeyPemToDer(
    const unsigned char * pem,
    int pemSz,
    unsigned char * buff,
    int buffSz,
```

```
    const char * pass
)
```

Converts a key in PEM format to DER format.

Parameters:

- **pem** a pointer to the PEM encoded certificate.
- **pemSz** the size of the PEM buffer (pem)
- **buff** a pointer to the copy of the buffer member of the DerBuffer struct.
- **buffSz** size of the buffer space allocated in the DerBuffer struct.
- **pass** password passed into the function.

See: wc_PemToDer

Return:

- int the function returns the number of bytes written to the buffer on successful execution.
- int negative int returned indicating an error.

Example

```
byte* loadBuf;
long fileSz = 0;
byte* bufSz;
static int LoadKeyFile(byte** keyBuf, word32* keyBufSz,
const char* keyFile,
                int typeKey, const char* password);
…
bufSz = wc_KeyPemToDer(loadBuf, (int)fileSz, saveBuf,
(int)fileSz, password);

if(saveBufSz > 0){
    // Bytes were written to the buffer.
}
```

C.5.2.50 function wc_CertPemToDer

```
int wc_CertPemToDer(
    const unsigned char * pem,
    int pemSz,
    unsigned char * buff,
    int buffSz,
    int type
)
```

This function converts a PEM formatted certificate to DER format. Calls OpenSSL function PemToDer.

Parameters:

- **pem** pointer PEM formatted certificate.
- **pemSz** size of the certificate.
- **buff** buffer to be copied to DER format.
- **buffSz** size of the buffer.
- **type** Certificate file type found in asn_public.h enum CertType.

See: wc_PemToDer

Return: buffer returns the bytes written to the buffer.

Example

```
const unsigned char* pem;
int pemSz;
unsigned char buff[BUFSIZE];
int buffSz = sizeof(buff)/sizeof(char);
int type;
...
if(wc_CertPemToDer(pem, pemSz, buff, buffSz, type) <= 0) {
    // There were bytes written to buffer
}
```

C.5.2.51 function wc_PemCertToDer_ex

```
int wc_PemCertToDer_ex(
    const char * fileName,
    DerBuffer ** der
)
```

Loads PEM certificate from file to DER buffer.

Parameters:

- **fileName** Path to PEM certificate file
- **der** Pointer to DerBuffer pointer to allocate

See:

- wc_CertPemToDer
- wc_FreeDer

Return:

- 0 on success
- negative on error

Example

```
DerBuffer* der = NULL;
int ret = wc_PemCertToDer_ex("cert.pem", &der);
if (ret == 0) {
    // Use der->buffer and der->length
    wc_FreeDer(&der);
}
```

C.5.2.52 function wc_PkcsPad

```
word32 wc_PkcsPad(
    byte * buf,
    word32 sz,
    word32 blockSz
)
```

Adds PKCS padding to buffer for RSA encryption.

Parameters:

- **buf** Buffer to pad
- **sz** Current size of data in buffer
- **blockSz** Block size for padding

See: wc_RsaPublicEncrypt

Return:

- Padded size on success
- 0 on error

Example

```
byte buffer[256];
word32 dataSz = 100;
word32 paddedSz = wc_PkcsPad(buffer, dataSz, 256);
```

C.5.2.53 function wc_RsaPublicKeyDecode_ex

```
int wc_RsaPublicKeyDecode_ex(
    const byte * input,
    word32 * inOutIdx,
    word32 inSz,
    const byte ** n,
    word32 * nSz,
    const byte ** e,
    word32 * eSz
)
```

Decodes RSA public key and extracts modulus and exponent.

Parameters:

- **input** DER encoded RSA public key buffer
- **inOutIdx** Pointer to index in buffer
- **inSz** Size of input buffer
- **n** Pointer to modulus pointer
- **nSz** Pointer to modulus size
- **e** Pointer to exponent pointer
- **eSz** Pointer to exponent size

See: wc_RsaPublicKeyDecode

Return:

- 0 on success
- negative on error

Example

```
const byte* n;
const byte* e;
word32 nSz, eSz, idx = 0;
int ret = wc_RsaPublicKeyDecode_ex(derBuf, &idx, derSz,
                                   &n, &nSz, &e, &eSz);
```

C.5.2.54 function wc_RsaPublicKeyDerSize

```
int wc_RsaPublicKeyDerSize(
    RsaKey * key,
    int with_header
)
```

Calculates DER encoded RSA public key size.

Parameters:

- **key** RSA key structure
- **with_header** Include sequence header if non-zero

See: wc_RsaKeyToDer

Return:

- Size on success
- negative on error

Example

```
RsaKey key;
int derSz = wc_RsaPublicKeyDerSize(&key, 1);
```

C.5.2.55 function wc_RsaPrivateKeyValidate

```
int wc_RsaPrivateKeyValidate(
    const byte * input,
    word32 * inOutIdx,
    int * keySz,
    word32 inSz
)
```

Validates DER encoded RSA private key format. This function validates the ASN.1 syntax and structure of the RSA private key (sequences, integer tags, and lengths) without loading the key values into an RsaKey structure. It does not perform mathematical validation of the RSA key parameters (e.g., checking if p and q are prime, or if the key components satisfy RSA mathematical relationships).

Parameters:

- **input** DER encoded RSA private key buffer
- **inOutIdx** Pointer to index in buffer (updated on success)
- **keySz** Pointer to store modulus size in bytes
- **inSz** Size of input buffer

See: wc_RsaPrivateKeyDecode

Return:

- 0 on success (valid ASN.1 structure)
- ASN_PARSE_E if ASN.1 parsing fails
- ASN_RSA_KEY_E if RSA key structure is invalid
- BAD_FUNC_ARG if parameters are invalid

Example

```
word32 idx = 0;
int keySz;
int ret = wc_RsaPrivateKeyValidate(derBuf, &idx, &keySz,
                                   derSz);
if (ret == 0) {
    // ASN.1 structure is valid, keySz contains modulus size
}
```

C.5.2.56 function wc_GetPubKeyDerFromCert

```
int wc_GetPubKeyDerFromCert(
    struct DecodedCert * cert,
    byte * derKey,
```

```
    word32 * derKeySz
)
```

This function gets the public key in DER format from a populated DecodedCert struct. Users must call wc_InitDecodedCert() and wc_ParseCert() before calling this API. wc_InitDecodedCert() accepts a DER/ASN.1 encoded certificate. To convert a PEM cert to DER, first use wc_CertPemToDer() before calling wc_InitDecodedCert().

Parameters:

- **cert** populated DecodedCert struct holding X.509 certificate
- **derKey** output buffer to place DER encoded public key
- **derKeySz** [IN/OUT] size of derKey buffer on input, size of public key on return. If derKey is passed in as NULL, derKeySz will be set to required buffer size for public key and LENGTH_ONLY_E will be returned from function.

See: wc_GetPubKeyDerFromCert

Return: 0 on success, negative on error. LENGTH_ONLY_E if derKey is NULL and returning length only.

C.5.2.57 function wc_DsaParamsDecode

```
int wc_DsaParamsDecode(
    const byte * input,
    word32 * inOutIdx,
    DsaKey * key,
    word32 inSz
)
```

Decodes DSA parameters from DER format.

Parameters:

- **input** DER encoded DSA parameters buffer
- **inOutIdx** Pointer to index in buffer
- **key** DSA key structure to store parameters
- **inSz** Size of input buffer

See: wc_DsaKeyToParamsDer

Return:

- 0 on success
- negative on error

Example

```
DsaKey key;
word32 idx = 0;
int ret = wc_DsaParamsDecode(derBuf, &idx, &key, derSz);
```

C.5.2.58 function wc_DsaKeyToParamsDer

```
int wc_DsaKeyToParamsDer(
    DsaKey * key,
    byte * output,
    word32 outLen
)
```

Encodes DSA parameters to DER format.

Parameters:

- **key** DSA key structure with parameters
- **output** Buffer for DER encoded parameters
- **outLen** Size of output buffer

See: wc_DsaParamsDecode

Return:

- Size on success
- negative on error

Example

```
DsaKey key;
byte der[1024];
int derSz = wc_DsaKeyToParamsDer(&key, der, sizeof(der));
```

C.5.2.59 function wc_DsaKeyToParamsDer_ex

```
int wc_DsaKeyToParamsDer_ex(
    DsaKey * key,
    byte * output,
    word32 * outLen
)
```

Encodes DSA parameters to DER with size output.

Parameters:

- **key** DSA key structure with parameters
- **output** Buffer for DER encoded parameters
- **outLen** Pointer to buffer size (in/out)

See: wc_DsaKeyToParamsDer

Return:

- 0 on success
- negative on error

Example

```
DsaKey key;
byte der[1024];
word32 derSz = sizeof(der);
int ret = wc_DsaKeyToParamsDer_ex(&key, der, &derSz);
```

C.5.2.60 function wc_DhParamsToDer

```
int wc_DhParamsToDer(
    DhKey * key,
    byte * out,
    word32 * outSz
)
```

Encodes DH parameters to DER format.

Parameters:

- **key** DH key structure with parameters
- **out** Buffer for DER encoded parameters
- **outSz** Pointer to buffer size (in/out)

See: wc_DhKeyToDer

Return:

- 0 on success
- negative on error

Example

```
DhKey key;
byte der[1024];
word32 derSz = sizeof(der);
int ret = wc_DhParamsToDer(&key, der, &derSz);
```

C.5.2.61 function wc_DhPubKeyToDer

```
int wc_DhPubKeyToDer(
    DhKey * key,
    byte * out,
    word32 * outSz
)
```

Encodes DH public key to DER format.

Parameters:

- **key** DH key structure with public key
- **out** Buffer for DER encoded public key
- **outSz** Pointer to buffer size (in/out)

See: wc_DhKeyToDer

Return:

- 0 on success
- negative on error

Example

```
DhKey key;
byte der[1024];
word32 derSz = sizeof(der);
int ret = wc_DhPubKeyToDer(&key, der, &derSz);
```

C.5.2.62 function wc_DhPrivKeyToDer

```
int wc_DhPrivKeyToDer(
    DhKey * key,
    byte * out,
    word32 * outSz
)
```

Encodes DH private key to DER format.

Parameters:

- **key** DH key structure with private key
- **out** Buffer for DER encoded private key
- **outSz** Pointer to buffer size (in/out)

See: wc_DhKeyToDer

Return:

- 0 on success
- negative on error

Example

```
DhKey key;
byte der[1024];
word32 derSz = sizeof(der);
int ret = wc_DhPrivKeyToDer(&key, der, &derSz);
```

C.5.2.63 function wc_EccPrivateKeyDecode

```
int wc_EccPrivateKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    ecc_key * key,
    word32 inSz
)
```

This function reads in an ECC private key from the input buffer, input, parses the private key, and uses it to generate an ecc_key object, which it stores in key.

Parameters:

- **input** pointer to the buffer containing the input private key
- **inOutIdx** pointer to a word32 object containing the index in the buffer at which to start
- **key** pointer to an initialized ecc object, on which to store the decoded private key
- **inSz** size of the input buffer containing the private key

See: wc_RSA_PrivateKeyDecode

Return:

- 0 On successfully decoding the private key and storing the result in the ecc_key struct
- ASN_PARSE_E: Returned if there is an error parsing the der file and storing it as a pem file
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- BUFFER_E Returned if the certificate to convert is large than the specified max certificate size
- ASN_OBJECT_ID_E Returned if the certificate encoding has an invalid object id
- ECC_CURVE_OID_E Returned if the ECC curve of the provided key is not supported
- ECC_BAD_ARG_E Returned if there is an error in the ECC key format
- NOT_COMPILED_IN Returned if the private key is compressed, and no compression key is provided
- MP_MEM Returned if there is an error in the math library used while parsing the private key
- MP_VAL Returned if there is an error in the math library used while parsing the private key
- MP_RANGE Returned if there is an error in the math library used while parsing the private key

Example

```
int ret, idx=0;
ecc_key key; // to store key in

byte* tmp; // tmp buffer to read key from
tmp = (byte*) malloc(FOURK_BUF);

int inSz;
inSz = fread(tmp, 1, FOURK_BUF, privateKeyFile);
// read key into tmp buffer

wc_ecc_init(&key); // initialize key
```

```
ret = wc_EccPrivateKeyDecode(tmp, &idx, &key, (word32)inSz);
if(ret < 0) {
    // error decoding ecc key
}
```

C.5.2.64 function wc_EccPrivateKeyToDer

```
int wc_EccPrivateKeyToDer(
    ecc_key * key,
    byte * output,
    word32 outLen
)
```

Encodes ECC private key to DER format.

Parameters:

- **key** ECC key structure with private key
- **output** Buffer for DER encoded private key
- **outLen** Size of output buffer

See: wc_EccPrivateKeyDecode

Return:

- Size on success
- negative on error

Example

```
ecc_key key;
byte der[1024];
int derSz = wc_EccPrivateKeyToDer(&key, der, sizeof(der));
```

C.5.2.65 function wc_EccKeyDerSize

```
int wc_EccKeyDerSize(
    ecc_key * key,
    int pub
)
```

Calculates DER encoded ECC key size.

Parameters:

- **key** ECC key structure
- **pub** Non-zero to include public key

See: wc_EccPrivateKeyToDer

Return:

- Size on success
- negative on error

Example

```
ecc_key key;
int derSz = wc_EccKeyDerSize(&key, 1);
```

C.5.2.66 function wc_EccPrivateKeyToPKCS8

```
int wc_EccPrivateKeyToPKCS8(
    ecc_key * key,
    byte * output,
    word32 * outLen
)
```

Encodes ECC private key to PKCS#8 format.

Parameters:

- **key** ECC key structure with private key
- **output** Buffer for PKCS#8 encoded key
- **outLen** Pointer to buffer size (in/out)

See: wc_EccPrivateKeyToDer

Return:

- Size on success
- negative on error

Example

```
ecc_key key;
byte pkcs8[1024];
word32 pkcs8Sz = sizeof(pkcs8);
int ret = wc_EccPrivateKeyToPKCS8(&key, pkcs8, &pkcs8Sz);
```

C.5.2.67 function wc_EccKeyToPKCS8

```
int wc_EccKeyToPKCS8(
    ecc_key * key,
    byte * output,
    word32 * outLen
)
```

Encodes ECC key pair to PKCS#8 format.

Parameters:

- **key** ECC key structure with key pair
- **output** Buffer for PKCS#8 encoded key
- **outLen** Pointer to buffer size (in/out)

See: wc_EccPrivateKeyToPKCS8

Return:

- Size on success
- negative on error

Example

```
ecc_key key;
byte pkcs8[1024];
word32 pkcs8Sz = sizeof(pkcs8);
int ret = wc_EccKeyToPKCS8(&key, pkcs8, &pkcs8Sz);
```

C.5.2.68 function wc_EccPublicKeyDerSize

```
int wc_EccPublicKeyDerSize(
    ecc_key * key,
    int with_AlgCurve
)
```

Calculates DER encoded ECC public key size.

Parameters:

- **key** ECC key structure
- **with_AlgCurve** Include algorithm and curve if non-zero

See: wc_EccPublicKeyToDer

Return:

- Size on success
- negative on error

Example

```
ecc_key key;
int derSz = wc_EccPublicKeyDerSize(&key, 1);
```

C.5.2.69 function wc_EccKeyToDer

```
int wc_EccKeyToDer(
    ecc_key * key,
    byte * output,
    word32 outLen
)
```

This function writes a private ECC key to der format.

Parameters:

- **key** pointer to the buffer containing the input ecc key
- **output** pointer to a buffer in which to store the der formatted key
- **outLen** the length of the buffer in which to store the der formatted key

See: wc_RsaKeyToDer

Return:

- Success On successfully writing the ECC key to der format, returns the length written to the buffer
- BAD_FUNC_ARG Returned if key or output is null, or outLen equals zero
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- BUFFER_E Returned if the converted certificate is too large to store in the output buffer
- ASN_UNKNOWN_OID_E Returned if the ECC key used is of an unknown type
- MP_MEM Returned if there is an error in the math library used while parsing the private key
- MP_VAL Returned if there is an error in the math library used while parsing the private key
- MP_RANGE Returned if there is an error in the math library used while parsing the private key

Example

```
int derSz;
ecc_key key;
// initialize and make key
byte der[FOURK_BUF];
// store der formatted key here
```

```
derSz = wc_EccKeyToDer(&key, der, FOURK_BUF);
if(derSz < 0) {
    // error converting ecc key to der buffer
}
```

C.5.2.70 function wc_EccPublicKeyDecode

```
int wc_EccPublicKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    ecc_key * key,
    word32 inSz
)
```

Decodes an ECC public key from an input buffer. It will parse an ASN sequence to retrieve the ECC key.

Parameters:

- **input** Buffer containing DER encoded key to decode.
- **inOutIdx** Index to start reading input buffer from. On output, index is set to last position parsed of input buffer.
- **key** Pointer to ecc_key struct to store the public key.
- **inSz** Size of the input buffer.

See: wc_ecc_import_x963

Return:

- 0 Success
- BAD_FUNC_ARG Returns if any arguments are null.
- ASN_PARSE_E Returns if there is an error parsing
- ASN_ECC_KEY_E Returns if there is an error importing the key. See wc_ecc_import_x963 for possible reasons.

Example

```
int ret;
word32 idx = 0;
byte buff[] = { // initialize with key };
ecc_key pubKey;
wc_ecc_init(&pubKey);
if ( wc_EccPublicKeyDecode(buff, &idx, &pubKey, sizeof(buff)) != 0) {
        // error decoding key
}
```

C.5.2.71 function wc_EccPublicKeyToDer

```
int wc_EccPublicKeyToDer(
    ecc_key * key,
    byte * output,
    word32 outLen,
    int with_AlgCurve
)
```

This function converts the ECC public key to DER format. It returns the size of buffer used. The public ECC key in DER format is stored in output buffer. The with_AlgCurve flag will include a header that has the Algorithm and Curve information.

Parameters:

- **key** Pointer to ECC key
- **output** Pointer to output buffer to write to.
- **outLen** Size of buffer.
- **with_AlgCurve** a flag for when to include a header that has the Algorithm and Curve information.

See:

- wc_EccKeyToDer
- wc_EccPrivateKeyDecode

Return:

- 0 Success, size of buffer used
- BAD_FUNC_ARG Returned if output or key is null.
- LENGTH_ONLY_E Error in getting ECC public key size.
- BUFFER_E Returned when output buffer is too small.

Example

```
ecc_key key;
wc_ecc_init(&key);
WC_RNG rng;
wc_InitRng(&rng);
wc_ecc_make_key(&rng, 32, &key);
int derSz = // Some appropriate size for der;
byte der[derSz];

if(wc_EccPublicKeyToDer(&key, der, derSz, 1) < 0)
{
    // Error converting ECC public key to der
}
```

C.5.2.72 function wc_EccPublicKeyToDer_ex

```
int wc_EccPublicKeyToDer_ex(
    ecc_key * key,
    byte * output,
    word32 outLen,
    int with_AlgCurve,
    int comp
)
```

This function converts the ECC public key to DER format. It returns the size of buffer used. The public ECC key in DER format is stored in output buffer. The with_AlgCurve flag will include a header that has the Algorithm and Curve information. The comp parameter determines if the public key will be exported as compressed.

Parameters:

- **key** Pointer to ECC key
- **output** Pointer to output buffer to write to.
- **outLen** Size of buffer.
- **with_AlgCurve** a flag for when to include a header that has the Algorithm and Curve information.
- **comp** If 1 (non-zero) the ECC public key will be written in compressed form. If 0 it will be written in an uncompressed format.

See:

- wc_EccKeyToDer
- wc_EccPublicKeyDecode

Return:

- 0 Success, size of buffer used
- BAD_FUNC_ARG Returned if output or key is null.
- LENGTH_ONLY_E Error in getting ECC public key size.
- BUFFER_E Returned when output buffer is too small.

Example

```
ecc_key key;
wc_ecc_init(&key);
WC_RNG rng;
wc_InitRng(&rng);
wc_ecc_make_key(&rng, 32, &key);
int derSz = // Some appropriate size for der;
byte der[derSz];

// Write out a compressed ECC key
if(wc_EccPublicKeyToDer_ex(&key, der, derSz, 1, 1) < 0)
{
    // Error converting ECC public key to der
}
```

C.5.2.73 function wc_Curve25519PrivateKeyDecode

```
int wc_Curve25519PrivateKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    curve25519_key * key,
    word32 inSz
)
```

This function decodes a Curve25519 private key (only) from a DER encoded buffer.

Parameters:

- **input** Pointer to buffer containing DER encoded private key
- **inOutIdx** Index to start reading input buffer from. On output, index is set to last position parsed of input buffer.
- **key** Pointer to curve25519_key structure to store decoded key
- **inSz** Size of input DER buffer

See:

- wc_Curve25519KeyDecode
- wc_Curve25519PublicKeyDecode

Return:

- 0 Success
- BAD_FUNC_ARG Returns if input, inOutIdx or key is null
- ASN_PARSE_E Returns if there is an error parsing the DER encoded data
- ECC_BAD_ARG_E Returns if the key length is not CURVE25519_KEYSIZE or the DER key contains other issues despite being properly formatted.
- BUFFER_E Returns if the input buffer is too small to contain a valid DER encoded key.

Example

```
byte der[] = { // DER encoded key };
word32 idx = 0;
curve25519_key key;
wc_curve25519_init(&key);

if (wc_Curve25519PrivateKeyDecode(der, &idx, &key, sizeof(der)) != 0) {
    // Error decoding private key
}
```

C.5.2.74 function wc_Curve25519PublicKeyDecode

```
int wc_Curve25519PublicKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    curve25519_key * key,
    word32 inSz
)
```

This function decodes a Curve25519 public key (only) from a DER encoded buffer.

Parameters:

- **input** Pointer to buffer containing DER encoded public key
- **inOutIdx** Index to start reading input buffer from. On output, index is set to last position parsed of input buffer.
- **key** Pointer to curve25519_key structure to store decoded key
- **inSz** Size of input DER buffer

See:

- wc_Curve25519KeyDecode
- wc_Curve25519PrivateKeyDecode

Return:

- 0 Success
- BAD_FUNC_ARG Returns if input, inOutIdx or key is null
- ASN_PARSE_E Returns if there is an error parsing the DER encoded data
- ECC_BAD_ARG_E Returns if the key length is not CURVE25519_KEYSIZE or the DER key contains other issues despite being properly formatted.
- BUFFER_E Returns if the input buffer is too small to contain a valid DER encoded key.

Example

```
byte der[] = { // DER encoded key };
word32 idx = 0;
curve25519_key key;
wc_curve25519_init(&key);
if (wc_Curve25519PublicKeyDecode(der, &idx, &key, sizeof(der)) != 0) {
    // Error decoding public key
}
```

C.5.2.75 function wc_Curve25519KeyDecode

```
int wc_Curve25519KeyDecode(
    const byte * input,
    word32 * inOutIdx,
```

```
    curve25519_key * key,
    word32 inSz
)
```

This function decodes a Curve25519 key from a DER encoded buffer. It can decode either a private key, a public key, or both.

Parameters:

- **input** Pointer to buffer containing DER encoded key
- **inOutIdx** Index to start reading input buffer from. On output, index is set to last position parsed of input buffer.
- **key** Pointer to curve25519_key structure to store decoded key
- **inSz** Size of input DER buffer

See:

- wc_Curve25519PrivateKeyDecode
- wc_Curve25519PublicKeyDecode

Return:

- 0 Success
- BAD_FUNC_ARG Returns if input, inOutIdx or key is null
- ASN_PARSE_E Returns if there is an error parsing the DER encoded data
- ECC_BAD_ARG_E Returns if the key length is not CURVE25519_KEYSIZE or the DER key contains other issues despite being properly formatted.
- BUFFER_E Returns if the input buffer is too small to contain a valid DER encoded key.

Example

```
byte der[] = { // DER encoded key };
word32 idx = 0;
curve25519_key key;
wc_curve25519_init(&key);
if (wc_Curve25519KeyDecode(der, &idx, &key, sizeof(der)) != 0) {
    // Error decoding key
}
```

C.5.2.76 function wc_Curve25519PrivateKeyToDer

```
int wc_Curve25519PrivateKeyToDer(
    curve25519_key * key,
    byte * output,
    word32 outLen
)
```

This function encodes a Curve25519 private key to DER format. If the input key structure contains a public key, it will be ignored.

Parameters:

- **key** Pointer to curve25519_key structure containing private key to encode
- **output** Buffer to hold DER encoding
- **outLen** Size of output buffer

See:

- wc_Curve25519KeyToDer
- wc_Curve25519PublicKeyToDer

Return:

- 0 Success, length of DER encoding
- BAD_FUNC_ARG Returns if key or output is null
- MEMORY_E Returns if there is an allocation failure
- BUFFER_E Returns if output buffer is too small

Example

```
curve25519_key key;
wc_curve25519_init(&key);
...
int derSz = 128; // Some appropriate size for output DER
byte der[derSz];
wc_Curve25519PrivateKeyToDer(&key, der, derSz);
```

C.5.2.77 function wc_Curve25519PublicKeyToDer

```
int wc_Curve25519PublicKeyToDer(
    curve25519_key * key,
    byte * output,
    word32 outLen,
    int withAlg
)
```

This function encodes a Curve25519 public key to DER format. If the input key structure contains a private key, it will be ignored.

Parameters:

- **key** Pointer to curve25519_key structure containing public key to encode
- **output** Buffer to hold DER encoding
- **outLen** Size of output buffer
- **withAlg** Whether to include algorithm identifier in the DER encoding

See:

- wc_Curve25519KeyToDer
- wc_Curve25519PrivateKeyToDer

Return:

- 0 Success, length of DER encoding
- BAD_FUNC_ARG Returns if key or output is null
- MEMORY_E Returns if there is an allocation failure
- BUFFER_E Returns if output buffer is too small

Example

```
curve25519_key key;
wc_curve25519_init(&key);
...
int derSz = 128; // Some appropriate size for output DER
byte der[derSz];
wc_Curve25519PublicKeyToDer(&key, der, derSz, 1);
```

C.5.2.78 function wc_Curve25519KeyToDer

```
int wc_Curve25519KeyToDer(
    curve25519_key * key,
    byte * output,
    word32 outLen,
    int withAlg
)
```

This function encodes a Curve25519 key to DER format. It can encode either a private key, a public key, or both.

Parameters:

- **key** Pointer to curve25519_key structure containing key to encode
- **output** Buffer to hold DER encoding
- **outLen** Size of output buffer
- **withAlg** Whether to include algorithm identifier in the DER encoding

See:

- wc_Curve25519PrivateKeyToDer
- wc_Curve25519PublicKeyToDer

Return:

- 0 Success, length of DER encoding
- BAD_FUNC_ARG Returns if key or output is null
- MEMORY_E Returns if there is an allocation failure
- BUFFER_E Returns if output buffer is too small

Example

```
curve25519_key key;
wc_curve25519_init(&key);
...
int derSz = 128; // Some appropriate size for output DER
byte der[derSz];
wc_Curve25519KeyToDer(&key, der, derSz, 1);
```

C.5.2.79 function wc_Ed25519PrivateKeyDecode

```
int wc_Ed25519PrivateKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    ed25519_key * key,
    word32 inSz
)
```

Decodes Ed25519 private key from DER format.

Parameters:

- **input** DER encoded Ed25519 private key buffer
- **inOutIdx** Pointer to index in buffer
- **key** Ed25519 key structure to store key
- **inSz** Size of input buffer

See:

- wc_Ed25519PrivateKeyToDer
- wc_Ed25519PrivateKeyDecode_ex

Return:

- 0 on success
- negative on error

Example

```
ed25519_key key;
word32 idx = 0;
int ret = wc_Ed25519PrivateKeyDecode(derBuf, &idx, &key,
                                     derSz);
```

C.5.2.80 function wc_Ed25519PrivateKeyDecode_ex

```
int wc_Ed25519PrivateKeyDecode_ex(
    const byte * input,
    word32 * inOutIdx,
    ed25519_key * key,
    word32 inSz,
    int trusted
)
```

Decodes Ed25519 private key from DER format with control over validation of a bundled public key. When the DER contains a public key and trusted is 0, the public key is validated with wc_ed25519_check_key(), including a check that it matches the private key. When trusted is 1 all validation of the bundled public key is skipped, so trusted should only be set with known_good key material. wc_Ed25519PrivateKeyDecode() behaves as trusted set to 0.

Parameters:

- **input** DER encoded Ed25519 private key buffer
- **inOutIdx** Pointer to index in buffer
- **key** Ed25519 key structure to store key
- **inSz** Size of input buffer
- **trusted** Indicates whether the bundled public key data is trusted. When 0, the public key is validated with wc_ed25519_check_key(). When 1, the public key is imported without any validation.

See:

- wc_Ed25519PrivateKeyDecode
- wc_Ed25519PrivateKeyToDer
- wc_ed25519_import_private_key_ex

Return:

- 0 on success
- negative on error

Example

```
ed25519_key key;
word32 idx = 0;
// key material comes from this application's own storage
int ret = wc_Ed25519PrivateKeyDecode_ex(derBuf, &idx, &key,
                                        derSz, 1);
```

C.5.2.81 function wc_Ed25519PublicKeyDecode

```
int wc_Ed25519PublicKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    ed25519_key * key,
    word32 inSz
)
```

Decodes Ed25519 public key from DER format.

Parameters:

- **input** DER encoded Ed25519 public key buffer
- **inOutIdx** Pointer to index in buffer
- **key** Ed25519 key structure to store key
- **inSz** Size of input buffer

See: wc_Ed25519PublicKeyToDer

Return:

- 0 on success
- negative on error

Example

```
ed25519_key key;
word32 idx = 0;
int ret = wc_Ed25519PublicKeyDecode(derBuf, &idx, &key,
                                    derSz);
```

C.5.2.82 function wc_Ed25519KeyToDer

```
int wc_Ed25519KeyToDer(
    const ed25519_key * key,
    byte * output,
    word32 outLen
)
```

Encodes Ed25519 key to DER format.

Parameters:

- **key** Ed25519 key structure
- **output** Buffer for DER encoded key
- **outLen** Size of output buffer

See: wc_Ed25519PrivateKeyToDer

Return:

- Size on success
- negative on error

Example

```
ed25519_key key;
byte der[1024];
int derSz = wc_Ed25519KeyToDer(&key, der, sizeof(der));
```

C.5.2.83 function wc_Ed25519PrivateKeyToDer

```
int wc_Ed25519PrivateKeyToDer(
    const ed25519_key * key,
    byte * output,
    word32 outLen
)
```

Encodes Ed25519 private key to DER format.

Parameters:

- **key** Ed25519 key structure with private key
- **output** Buffer for DER encoded private key
- **outLen** Size of output buffer

See: wc_Ed25519PrivateKeyDecode

Return:

- Size on success
- negative on error

Example

```
ed25519_key key;
byte der[1024];
int derSz = wc_Ed25519PrivateKeyToDer(&key, der,
                                      sizeof(der));
```

C.5.2.84 function wc_Ed25519PublicKeyToDer

```
int wc_Ed25519PublicKeyToDer(
    const ed25519_key * key,
    byte * output,
    word32 outLen,
    int withAlg
)
```

Encodes Ed25519 public key to DER format.

Parameters:

- **key** Ed25519 key structure with public key
- **output** Buffer for DER encoded public key
- **outLen** Size of output buffer
- **withAlg** Whether to use SubjectPublicKeyInfo format.

See: wc_Ed25519PublicKeyDecode

Return:

- Size on success
- negative on error

Example

```
ed25519_key key;
byte der[1024];
int derSz = wc_Ed25519PublicKeyToDer(&key, der,
                                     sizeof(der));
```

C.5.2.85 function wc_Ed448PrivateKeyDecode

```
int wc_Ed448PrivateKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    ed448_key * key,
    word32 inSz
)
```

Decodes Ed448 private key from DER format.

Parameters:

- **input** DER encoded Ed448 private key buffer
- **inOutIdx** Pointer to index in buffer
- **key** Ed448 key structure to store key
- **inSz** Size of input buffer

See:

- wc_Ed448PrivateKeyToDer
- wc_Ed448PrivateKeyDecode_ex

Return:

- 0 on success
- negative on error

Example

```
ed448_key key;
word32 idx = 0;
int ret = wc_Ed448PrivateKeyDecode(derBuf, &idx, &key,
                                   derSz);
```

C.5.2.86 function wc_Ed448PrivateKeyDecode_ex

```
int wc_Ed448PrivateKeyDecode_ex(
    const byte * input,
    word32 * inOutIdx,
    ed448_key * key,
    word32 inSz,
    int trusted
)
```

Decodes Ed448 private key from DER format with control over validation of a bundled public key. When the DER contains a public key and trusted is 0, the public key is validated with wc_ed448_check_key(), including a check that it matches the private key. When trusted is 1 all validation of the bundled public key is skipped, so trusted should only be set with known_good key material. wc_Ed448PrivateKeyDecode() behaves as trusted set to 0.

Parameters:

- **input** DER encoded Ed448 private key buffer
- **inOutIdx** Pointer to index in buffer
- **key** Ed448 key structure to store key
- **inSz** Size of input buffer
- **trusted** Indicates whether the bundled public key data is trusted. When 0, the public key is validated with wc_ed448_check_key(). When 1, the public key is imported without any validation.

See:

- wc_Ed448PrivateKeyDecode
- wc_Ed448PrivateKeyToDer
- wc_ed448_import_private_key_ex

Return:

- 0 on success
- negative on error

Example

```
ed448_key key;
word32 idx = 0;
// key material comes from this application's own storage
int ret = wc_Ed448PrivateKeyDecode_ex(derBuf, &idx, &key,
                                      derSz, 1);
```

C.5.2.87 function wc_Ed448PublicKeyDecode

```
int wc_Ed448PublicKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    ed448_key * key,
    word32 inSz
)
```

Decodes Ed448 public key from DER format.

Parameters:

- **input** DER encoded Ed448 public key buffer
- **inOutIdx** Pointer to index in buffer
- **key** Ed448 key structure to store key
- **inSz** Size of input buffer

See: wc_Ed448PublicKeyToDer

Return:

- 0 on success
- negative on error

Example

```
ed448_key key;
word32 idx = 0;
int ret = wc_Ed448PublicKeyDecode(derBuf, &idx, &key,
                                  derSz);
```

C.5.2.88 function wc_Ed448KeyToDer

```
int wc_Ed448KeyToDer(
    const ed448_key * key,
    byte * output,
    word32 outLen
)
```

Encodes Ed448 key to DER format.

Parameters:

- **key** Ed448 key structure

- **output** Buffer for DER encoded key
- **outLen** Size of output buffer

See: wc_Ed448PrivateKeyToDer

Return:

- Size on success
- negative on error

Example

```
ed448_key key;
byte der[1024];
int derSz = wc_Ed448KeyToDer(&key, der, sizeof(der));
```

C.5.2.89 function wc_Ed448PrivateKeyToDer

```
int wc_Ed448PrivateKeyToDer(
    const ed448_key * key,
    byte * output,
    word32 outLen
)
```

Encodes Ed448 private key to DER format.

Parameters:

- **key** Ed448 key structure with private key
- **output** Buffer for DER encoded private key
- **outLen** Size of output buffer

See: wc_Ed448PrivateKeyDecode

Return:

- Size on success
- negative on error

Example

```
ed448_key key;
byte der[1024];
int derSz = wc_Ed448PrivateKeyToDer(&key, der,
                                    sizeof(der));
```

C.5.2.90 function wc_Ed448PublicKeyToDer

```
int wc_Ed448PublicKeyToDer(
    const ed448_key * key,
    byte * output,
    word32 outLen,
    int withAlg
)
```

Encodes Ed448 public key to DER format.

Parameters:

- **key** Ed448 key structure with public key
- **output** Buffer for DER encoded public key
- **outLen** Size of output buffer

- **withAlg** 1 to include algorithm identifier, 0 for key data only

See: wc_Ed448PublicKeyDecode

Return:

- Size on success
- negative on error

Example

```
ed448_key key;
byte der[1024];
int derSz = wc_Ed448PublicKeyToDer(&key, der,
                                   sizeof(der), 1);
```

C.5.2.91 function wc_Curve448PrivateKeyDecode

```
int wc_Curve448PrivateKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    curve448_key * key,
    word32 inSz
)
```

Decodes Curve448 private key from DER format.

Parameters:

- **input** DER encoded Curve448 private key buffer
- **inOutIdx** Pointer to index in buffer
- **key** Curve448 key structure to store key
- **inSz** Size of input buffer

See: wc_Curve448PrivateKeyToDer

Return:

- 0 on success
- negative on error

Example

```
curve448_key key;
word32 idx = 0;
int ret = wc_Curve448PrivateKeyDecode(derBuf, &idx, &key,
                                      derSz);
```

C.5.2.92 function wc_Curve448PublicKeyDecode

```
int wc_Curve448PublicKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    curve448_key * key,
    word32 inSz
)
```

Decodes Curve448 public key from DER format.

Parameters:

- **input** DER encoded Curve448 public key buffer

- **inOutIdx** Pointer to index in buffer
- **key** Curve448 key structure to store key
- **inSz** Size of input buffer

See: wc_Curve448PublicKeyToDer

Return:

- 0 on success
- negative on error

Example

```
curve448_key key;
word32 idx = 0;
int ret = wc_Curve448PublicKeyDecode(derBuf, &idx, &key,
                                     derSz);
```

C.5.2.93 function wc_Curve448PrivateKeyToDer

```
int wc_Curve448PrivateKeyToDer(
    curve448_key * key,
    byte * output,
    word32 outLen
)
```

Encodes Curve448 private key to DER format.

Parameters:

- **key** Curve448 key structure with private key
- **output** Buffer for DER encoded private key
- **outLen** Size of output buffer

See: wc_Curve448PrivateKeyDecode

Return:

- Size on success
- negative on error

Example

```
curve448_key key;
byte der[1024];
int derSz = wc_Curve448PrivateKeyToDer(&key, der,
                                       sizeof(der));
```

C.5.2.94 function wc_Curve448PublicKeyToDer

```
int wc_Curve448PublicKeyToDer(
    curve448_key * key,
    byte * output,
    word32 outLen
)
```

Encodes Curve448 public key to DER format.

Parameters:

- **key** Curve448 key structure with public key
- **output** Buffer for DER encoded public key

- **outLen** Size of output buffer

See: wc_Curve448PublicKeyDecode

Return:

- Size on success
- negative on error

Example

```
curve448_key key;
byte der[1024];
int derSz = wc_Curve448PublicKeyToDer(&key, der,
                                      sizeof(der));
```

C.5.2.95 function wc_EncodeSignature

```
word32 wc_EncodeSignature(
    byte * out,
    const byte * digest,
    word32 digSz,
    int hashOID
)
```

This function encodes a digital signature into the output buffer, and returns the size of the encoded signature created.

Parameters:

- **out** pointer to the buffer where the encoded signature will be written
- **digest** pointer to the digest to use to encode the signature
- **digSz** the length of the buffer containing the digest
- **hashOID** OID identifying the hash type used to generate the signature. Valid options, depending on build configurations, are: SHAh, SHA256h, SHA384h, SHA512h, MD2h, MD5h, DESb, DES3b, CTC_MD5wRSA, CTC_SHAwRSA, CTC_SHA256wRSA, CTC_SHA384wRSA, CTC_SHA512wRSA, CTC_SHAwECDSA, CTC_SHA256wECDSA, CTC_SHA384wECDSA, and CTC_SHA512wECDSA.

See: none

Return: Success On successfully writing the encoded signature to output, returns the length written to the buffer

```
int signSz;
byte encodedSig[MAX_ENCODED_SIG_SZ];
Sha256 sha256;
// initialize sha256 for hashing

byte* dig = = (byte*)malloc(WC_SHA256_DIGEST_SIZE);
// perform hashing and hash updating so dig stores SHA-256 hash
// (see wc_InitSha256, wc_Sha256Update and wc_Sha256Final)
signSz = wc_EncodeSignature(encodedSig, dig, WC_SHA256_DIGEST_SIZE, SHA256h);
```

C.5.2.96 function wc_GetCTC_HashOID

```
int wc_GetCTC_HashOID(
    int type
)
```

This function returns the hash OID that corresponds to a hashing type. For example, when given the type: WC_SHA512, this function returns the identifier corresponding to a SHA512 hash, SHA512h.

Parameters:

- **type** the hash type for which to find the OID. Valid options, depending on build configuration, include: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512

See: none

Return:

- Success On success, returns the OID corresponding to the appropriate hash to use with that encryption type.
- 0 Returned if an unrecognized hash type is passed in as argument.

Example

```
int hashOID;

hashOID = wc_GetCTC_HashOID(WC_SHA512);
if (hashOID == 0) {
    // WOLFSSL_SHA512 not defined
}
```

C.5.2.97 function wc_SetCert_Free

```
void wc_SetCert_Free(
    Cert * cert
)
```

This function cleans up memory and resources used by the certificate structure's decoded cert cache. When WOLFSSL_CERT_GEN_CACHE is defined the decoded cert structure is cached in the certificate structure. This allows subsequent calls to certificate set functions to avoid parsing the decoded cert on each call.

Parameters:

- **cert** pointer to an uninitialized certificate information structure.

See:

- wc_SetAuthKeyIdFromCert
- wc_SetIssuerBuffer
- wc_SetSubjectBuffer
- wc_SetSubjectRaw
- wc_SetIssuerRaw
- wc_SetAltNamesBuffer
- wc_SetDatesBuffer

Return:

- 0 on success.
- BAD_FUNC_ARG Returned if invalid pointer is passed in as argument.

Example

```
Cert cert; // Initialized certificate structure

wc_SetCert_Free(&cert);
```

C.5.2.98 function wc_GetPkcs8TraditionalOffset

```
int wc_GetPkcs8TraditionalOffset(
    byte * input,
    word32 * inOutIdx,
    word32 sz
)
```

This function finds the beginning of the traditional private key inside a PKCS#8 unencrypted buffer.

Parameters:

- **input** Buffer containing unencrypted PKCS#8 private key.
- **inOutIdx** Index into the input buffer. On input, it should be a byte offset to the beginning of the the PKCS#8 buffer. On output, it will be the byte offset to the traditional private key within the input buffer.
- **sz** The number of bytes in the input buffer.

See:

- wc_CreatePKCS8Key
- wc_EncryptPKCS8Key
- wc_DecryptPKCS8Key
- wc_CreateEncryptedPKCS8Key

Return:

- Length of traditional private key on success.
- Negative values on failure.

Example

```
byte* pkcs8Buf; // Buffer containing PKCS#8 key.
word32 idx = 0;
word32 sz; // Size of pkcs8Buf.
...
ret = wc_GetPkcs8TraditionalOffset(pkcs8Buf, &idx, sz);
// pkcs8Buf + idx is now the beginning of the traditional private key bytes.
```

C.5.2.99 function wc_CreatePKCS8Key

```
int wc_CreatePKCS8Key(
    byte * out,
    word32 * outSz,
    byte * key,
    word32 keySz,
    int algoID,
    const byte * curveOID,
    word32 oidSz
)
```

This function takes in a DER private key and converts it to PKCS#8 format. Also used in creating PKCS#12 shrouded key bags. See RFC 5208.

Parameters:

- **out** Buffer to place result in. If NULL, required out buffer size returned in outSz.
- **outSz** Size of out buffer.
- **key** Buffer with traditional DER key.
- **keySz** Size of key buffer.
- **algoID** Algorithm ID (e.g. RSAk).

- **curveOID** ECC curve OID if used. Should be NULL for RSA keys.
- **oidSz** Size of curve OID. Is set to 0 if curveOID is NULL.

See:

- wc_GetPkcs8TraditionalOffset
- wc_EncryptPKCS8Key
- wc_DecryptPKCS8Key
- wc_CreateEncryptedPKCS8Key

Return:

- The size of the PKCS#8 key placed into out on success.
- LENGTH_ONLY_E if out is NULL, with required output buffer size in outSz.
- Other negative values on failure.

Example

```
ecc_key eccKey;              // wolfSSL ECC key object.
byte* der;                   // DER-encoded ECC key.
word32 derSize;              // Size of der.
const byte* curveOid = NULL; // OID of curve used by eccKey.
word32 curveOidSz = 0;       // Size of curve OID.
byte* pkcs8;                 // Output buffer for PKCS#8 key.
word32 pkcs8Sz;              // Size of output buffer.

derSize = wc_EccKeyDerSize(&eccKey, 1);
...
derSize = wc_EccKeyToDer(&eccKey, der, derSize);
...
ret = wc_ecc_get_oid(eccKey.dp->oidSum, &curveOid, &curveOidSz);
...
ret = wc_CreatePKCS8Key(NULL, &pkcs8Sz, der,
    derSize, ECDSAk, curveOid, curveOidSz); // Get size needed in pkcs8Sz.
...
ret = wc_CreatePKCS8Key(pkcs8, &pkcs8Sz, der,
    derSize, ECDSAk, curveOid, curveOidSz);
```

C.5.2.100 function wc_EncryptPKCS8Key

```
int wc_EncryptPKCS8Key(
    byte * key,
    word32 keySz,
    byte * out,
    word32 * outSz,
    const char * password,
    int passwordSz,
    int vPKCS,
    int pbeOid,
    int encAlgId,
    byte * salt,
    word32 saltSz,
    int itt,
    WC_RNG * rng,
    void * heap
)
```

This function takes in an unencrypted PKCS#8 DER key (e.g. one created by wc_CreatePKCS8Key) and converts it to PKCS#8 encrypted format. The resulting encrypted key can be decrypted using wc_DecryptPKCS8Key. See RFC 5208.

Parameters:

- **key** Buffer with traditional DER key.
- **keySz** Size of key buffer.
- **out** Buffer to place result in. If NULL, required out buffer size returned in outSz.
- **outSz** Size of out buffer.
- **password** The password to use for the password-based encryption algorithm.
- **passwordSz** The length of the password (not including the NULL terminator).
- **vPKCS** The PKCS version to use. Can be 1 for PKCS12 or PKCS5.
- **pbeOid** The OID of the PBE scheme to use (e.g. PBES2 or one of the OIDs for PBES1 in RFC 2898 A.3).
- **encAlgId** The encryption algorithm ID to use (e.g. AES256CBCb).
- **salt** The salt buffer to use. If NULL, a random salt will be used.
- **saltSz** The length of the salt buffer. Can be 0 if passing NULL for salt.
- **itt** The number of iterations to use for the KDF.
- **rng** A pointer to an initialized WC_RNG object.
- **heap** A pointer to the heap used for dynamic allocation. Can be NULL.

See:

- wc_GetPkcs8TraditionalOffset
- wc_CreatePKCS8Key
- wc_DecryptPKCS8Key
- wc_CreateEncryptedPKCS8Key

Return:

- The size of the encrypted key placed in out on success.
- LENGTH_ONLY_E if out is NULL, with required output buffer size in outSz.
- Other negative values on failure.

Example

```
byte* pkcs8;          // Unencrypted PKCS#8 key.
word32 pkcs8Sz;       // Size of pkcs8.
byte* pkcs8Enc;       // Encrypted PKCS#8 key.
word32 pkcs8EncSz;    // Size of pkcs8Enc.
const char* password; // Password to use for encryption.
int passwordSz;       // Length of password (not including NULL terminator).
WC_RNG rng;

// The following produces an encrypted version of pkcs8 in pkcs8Enc. The
// encryption uses password-based encryption scheme 2 (PBE2) from PKCS#5 and
// the AES cipher in CBC mode with a 256-bit key. See RFC 8018 for more on
// PKCS#5.
ret = wc_EncryptPKCS8Key(pkcs8, pkcs8Sz, pkcs8Enc, &pkcs8EncSz, password,
        passwordSz, PKCS5, PBES2, AES256CBCb, NULL, 0,
        WC_PKCS12_ITT_DEFAULT, &rng, NULL);
```

C.5.2.101 function wc_EncryptPKCS8Key_ex

```
int wc_EncryptPKCS8Key_ex(
    byte * key,
    word32 keySz,
```

```
    byte * out,
    word32 * outSz,
    const char * password,
    int passwordSz,
    int vPKCS,
    int pbeOid,
    int encAlgId,
    byte * salt,
    word32 saltSz,
    int itt,
    WC_RNG * rng,
    void * heap,
    int devId
)
```

Encrypts PKCS#8 key with extended parameters.

Parameters:

- **key** Private key buffer
- **keySz** Size of private key
- **out** Output buffer for encrypted key
- **outSz** Pointer to output buffer size (in/out)
- **password** Password for encryption
- **passwordSz** Password length
- **vPKCS** PKCS version
- **pbeOid** PBE algorithm OID
- **encAlgId** Encryption algorithm ID
- **salt** Salt buffer
- **saltSz** Salt size
- **itt** Iteration count
- **rng** Random number generator
- **heap** Heap hint for memory allocation
- **devId** Device ID for hardware acceleration

See: wc_EncryptPKCS8Key

Return:

- Size on success
- negative on error

Example

```
byte key[256], encrypted[512];
word32 encSz = sizeof(encrypted);
WC_RNG rng;
int ret = wc_EncryptPKCS8Key_ex(key, keySz, encrypted,
                                &encSz, "password", 8,
                                PKCS5, PBES2, AES256CBCb,
                                NULL, 0, 2048, &rng, NULL,
                                INVALID_DEVID);
```

C.5.2.102 function wc_GetTime

```
int wc_GetTime(
    void * timePtr,
```

```
    word32 timeSize
)
```

Gets current time for certificate operations.

Parameters:

- **timePtr** Pointer to time buffer
- **timeSize** Size of time buffer

See: wc_GetDateInfo

Return:

- 0 on success
- negative on error

Example

```
time_t currentTime;
int ret = wc_GetTime(&currentTime, sizeof(currentTime));
```

C.5.2.103 function wc_EncryptedInfoGet

```
int wc_EncryptedInfoGet(
    EncryptedInfo * info,
    const char * cipherName
)
```

Gets encryption info from encrypted PEM.

Parameters:

- **info** EncryptedInfo structure to populate
- **cipherName** Cipher name string

See: wc_PemToDer

Return:

- 0 on success
- negative on error

Example

```
EncryptedInfo info;
int ret = wc_EncryptedInfoGet(&info, "AES-256-CBC");
```

C.5.2.104 function wc_ParseCertPIV

```
int wc_ParseCertPIV(
    wc_CertPIV * cert,
    const byte * buf,
    word32 totalSz
)
```

Parses PIV certificate format.

Parameters:

- **cert** PIV certificate structure to populate
- **buf** Buffer containing PIV certificate
- **totalSz** Size of buffer

See: wc_InitDecodedCert

Return:

- 0 on success
- negative on error

Example

```
wc_CertPIV cert;
int ret = wc_ParseCertPIV(&cert, pivBuf, pivSz);
```

C.5.2.105 function wc_GetSubjectPubKeyInfoDerFromCert

```
int wc_GetSubjectPubKeyInfoDerFromCert(
    const byte * certDer,
    word32 certDerSz,
    byte * pubKeyDer,
    word32 * pubKeyDerSz
)
```

Extracts subject public key info from certificate.

Parameters:

- **certDer** DER encoded certificate buffer
- **certDerSz** Size of certificate
- **pubKeyDer** Output buffer for public key
- **pubKeyDerSz** Pointer to output buffer size (in/out)

See: wc_GetPubKeyDerFromCert

Return:

- Size on success
- negative on error

Example

```
byte pubKey[1024];
word32 pubKeySz = sizeof(pubKey);
int ret = wc_GetSubjectPubKeyInfoDerFromCert(certDer,
                                             certSz,
                                             pubKey,
                                             &pubKeySz);
```

C.5.2.106 function wc_GetDecodedCertSubject

```
int wc_GetDecodedCertSubject(
    const struct DecodedCert * cert,
    char * buf,
    word32 * bufSz
)
```

Retrieves the subject name from a decoded certificate.

Parameters:

- **cert** Pointer to the DecodedCert (must have been parsed).
- **buf** Output buffer to receive the subject name string, or NULL to query the required size.
- **bufSz** Pointer to the buffer size. On input, the available buffer size. On output, the number of bytes written (excluding any NUL terminator) or the required size if buf is NULL.

See:

- wc_GetDecodedCertIssuer
- wc_GetDecodedCertSerial
- wc_InitDecodedCert
- wc_ParseCert

Return:

- 0 on success.
- LENGTH_ONLY_E when buf is NULL (bufSz contains required size).
- BAD_FUNC_ARG if cert or bufSz is NULL.
- BUFFER_E if the provided buffer is too small.

This function copies the subject name string from a DecodedCert structure into the provided buffer. The string uses a one-line distinguished name format with "/" delimiters (e.g. "/C=US/O=Org/CN=example.com"). The output is NOT NUL-terminated; the caller should append a NUL byte if needed. If buf is NULL, the required buffer size is returned in bufSz and LENGTH_ONLY_E is returned.

C.5.2.107 function wc_GetDecodedCertIssuer

```
int wc_GetDecodedCertIssuer(
    const struct DecodedCert * cert,
    char * buf,
    word32 * bufSz
)
```

Retrieves the issuer name from a decoded certificate.

Parameters:

- **cert** Pointer to the DecodedCert (must have been parsed).
- **buf** Output buffer to receive the issuer name string, or NULL to query the required size.
- **bufSz** Pointer to the buffer size. On input, the available buffer size. On output, the number of bytes written (excluding any NUL terminator) or the required size if buf is NULL.

See:

- wc_GetDecodedCertSubject
- wc_GetDecodedCertSerial
- wc_InitDecodedCert
- wc_ParseCert

Return:

- 0 on success.
- LENGTH_ONLY_E when buf is NULL (bufSz contains required size).
- BAD_FUNC_ARG if cert or bufSz is NULL.
- BUFFER_E if the provided buffer is too small.

This function copies the issuer name string from a DecodedCert structure into the provided buffer. The string uses a one-line distinguished name format with "/" delimiters (e.g. "/C=US/O=Org/CN=example.com"). The output is NOT NUL-terminated; the caller should append a NUL byte if needed. If buf is NULL, the required buffer size is returned in bufSz and LENGTH_ONLY_E is returned.

C.5.2.108 function wc_GetDecodedCertSerial

```
int wc_GetDecodedCertSerial(
    const struct DecodedCert * cert,
    byte * buf,
```

```
    word32 * bufSz
)
```

Retrieves the serial number from a decoded certificate.

Parameters:

- **cert** Pointer to the DecodedCert (must have been parsed).
- **buf** Output buffer to receive the serial number bytes, or NULL to query the required size.
- **bufSz** Pointer to the buffer size. On input, the available buffer size. On output, the number of bytes written or the required size if buf is NULL.

See:

- wc_GetDecodedCertSubject
- wc_GetDecodedCertIssuer
- wc_InitDecodedCert
- wc_ParseCert

Return:

- 0 on success.
- LENGTH_ONLY_E when buf is NULL (bufSz contains required size).
- BAD_FUNC_ARG if cert or bufSz is NULL.
- BUFFER_E if the provided buffer is too small.

This function copies the serial number bytes from a DecodedCert structure into the provided buffer. If buf is NULL, the required buffer size is returned in bufSz and LENGTH_ONLY_E is returned.

C.5.2.109 function wc_GetUUIDFromCert

```
int wc_GetUUIDFromCert(
    struct DecodedCert * cert,
    byte * uuid,
    int * uuidSz
)
```

Extracts UUID from certificate.

Parameters:

- **cert** Decoded certificate structure
- **uuid** Output buffer for UUID
- **uuidSz** Pointer to UUID buffer size (in/out)

See: wc_ParseCert

Return:

- 0 on success
- negative on error

Example

```
DecodedCert cert;
byte uuid[16];
int uuidSz = sizeof(uuid);
int ret = wc_GetUUIDFromCert(&cert, uuid, &uuidSz);
```

C.5.2.110 function wc_GetFASCNFromCert

```
int wc_GetFASCNFromCert(
    struct DecodedCert * cert,
    byte * fascn,
    int * fascnSz
)
```

Extracts FASCN from certificate.

Parameters:

- **cert** Decoded certificate structure
- **fascn** Output buffer for FASCN
- **fascnSz** Pointer to FASCN buffer size (in/out)

See: wc_ParseCert

Return:

- 0 on success
- negative on error

Example

```
DecodedCert cert;
byte fascn[25];
int fascnSz = sizeof(fascn);
int ret = wc_GetFASCNFromCert(&cert, fascn, &fascnSz);
```

C.5.2.111 function wc_GeneratePreTBS

```
int wc_GeneratePreTBS(
    struct DecodedCert * cert,
    byte * der,
    int derSz
)
```

Generates the pre-TBS (To Be Signed) certificate data from a decoded certificate. The TBS portion is the certificate data that gets signed by the certificate authority. This function is used in dual algorithm certificate creation where the TBS data needs to be extracted for signing with an alternative algorithm (e.g., a post-quantum algorithm).

Parameters:

- **cert** Decoded certificate structure containing the certificate to extract TBS data from
- **der** Output buffer for the pre-TBS DER-encoded data
- **derSz** Size of output buffer in bytes

See:

- wc_MakeCert
- wc_MakeSigWithBitStr

Return:

- Size of the pre-TBS data on success
- Negative error code on failure

Note: This API is only available when WOLFSSL_DUAL_ALG_CERTS is defined, which enables support for dual algorithm certificates used in Post-Quantum cryptography to provide hybrid signing with both traditional and PQ algorithms.

Example

```
DecodedCert cert;
byte preTbs[2048];
int ret = wc_GeneratePreTBS(&cert, preTbs, sizeof(preTbs));
if (ret > 0) {
    // ret contains the size of the pre-TBS data
    // preTbs can now be signed with an alternative algorithm
}
```

C.5.2.112 function wc_InitDecodedAcert

```
void wc_InitDecodedAcert(
    struct DecodedAcert * acert,
    void * heap
)
```

Initializes decoded attribute certificate structure.

Parameters:

- **acert** Attribute certificate structure to initialize
- **heap** Heap hint for memory allocation

See: wc_FreeDecodedAcert

Return: void

Example

```
DecodedAcert acert;
wc_InitDecodedAcert(&acert, NULL);
```

C.5.2.113 function wc_FreeDecodedAcert

```
void wc_FreeDecodedAcert(
    struct DecodedAcert * acert
)
```

Frees decoded attribute certificate structure.

Parameters:

- **acert** Attribute certificate structure to free

See: wc_InitDecodedAcert

Return: void

Example

```
DecodedAcert acert;
wc_InitDecodedAcert(&acert, NULL);
wc_FreeDecodedAcert(&acert);
```

C.5.2.114 function wc_ParseX509Acert

```
int wc_ParseX509Acert(
    struct DecodedAcert * acert,
    int verify
)
```

Parses X.509 attribute certificate.

Parameters:

- **acert** Decoded attribute certificate structure
- **verify** Non-zero to verify signature

See: wc_VerifyX509Acert

Return:

- 0 on success
- negative on error

Example

```
DecodedAcert acert;
wc_InitDecodedAcert(&acert, NULL);
int ret = wc_ParseX509Acert(&acert, 1);
```

C.5.2.115 function wc_VerifyX509Acert

```
int wc_VerifyX509Acert(
    const byte * acert,
    word32 acertSz,
    const byte * issuerCert,
    word32 issuerCertSz,
    void * cm
)
```

Verifies X.509 attribute certificate.

Parameters:

- **acert** Attribute certificate buffer
- **acertSz** Size of attribute certificate
- **issuerCert** Issuer certificate buffer
- **issuerCertSz** Size of issuer certificate
- **cm** Certificate manager

See: wc_ParseX509Acert

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_VerifyX509Acert(acertBuf, acertSz,
                             issuerBuf, issuerSz, cm);
```

C.5.2.116 function wc_DecryptPKCS8Key

```
int wc_DecryptPKCS8Key(
    byte * input,
    word32 sz,
    const char * password,
    int passwordSz
)
```

This function takes an encrypted PKCS#8 DER key and decrypts it to PKCS#8 unencrypted DER.Undoes the encryption done by wc_EncryptPKCS8Key. See RFC5208. The input buffer is overwritten with the decrypted data.

Parameters:

- **input** On input, buffer containing encrypted PKCS#8 key. On successful output, contains the decrypted key.
- **sz** Size of the input buffer.
- **password** The password used to encrypt the key.
- **passwordSz** The length of the password (not including NULL terminator).

See:

- wc_GetPkcs8TraditionalOffset
- wc_CreatePKCS8Key
- wc_EncryptPKCS8Key
- wc_CreateEncryptedPKCS8Key

Return:

- The length of the decrypted buffer on success.
- Negative values on failure.

Example

```
byte* pkcs8Enc;       // Encrypted PKCS#8 key made with wc_EncryptPKCS8Key.
word32 pkcs8EncSz;    // Size of pkcs8Enc.
const char* password; // Password to use for decryption.
int passwordSz;       // Length of password (not including NULL terminator).

ret = wc_DecryptPKCS8Key(pkcs8Enc, pkcs8EncSz, password, passwordSz);
```

C.5.2.117 function wc_CreateEncryptedPKCS8Key

```
int wc_CreateEncryptedPKCS8Key(
    byte * key,
    word32 keySz,
    byte * out,
    word32 * outSz,
    const char * password,
    int passwordSz,
    int vPKCS,
    int pbeOid,
    int encAlgId,
    byte * salt,
    word32 saltSz,
    int itt,
    WC_RNG * rng,
    void * heap
)
```

This function takes a traditional, DER key, converts it to PKCS#8 format, and encrypts it. It uses wc_CreatePKCS8Key and wc_EncryptPKCS8Key to do this.

Parameters:

- **key** Buffer with traditional DER key.
- **keySz** Size of key buffer.
- **out** Buffer to place result in. If NULL, required out buffer size returned in outSz.

- **outSz** Size of out buffer.
- **password** The password to use for the password-based encryption algorithm.
- **passwordSz** The length of the password (not including the NULL terminator).
- **vPKCS** The PKCS version to use. Can be 1 for PKCS12 or PKCS5.
- **pbeOid** The OID of the PBE scheme to use (e.g. PBES2 or one of the OIDs for PBES1 in RFC 2898 A.3).
- **encAlgId** The encryption algorithm ID to use (e.g. AES256CBCb).
- **salt** The salt buffer to use. If NULL, a random salt will be used.
- **saltSz** The length of the salt buffer. Can be 0 if passing NULL for salt.
- **itt** The number of iterations to use for the KDF.
- **rng** A pointer to an initialized WC_RNG object.
- **heap** A pointer to the heap used for dynamic allocation. Can be NULL.

See:

- wc_GetPkcs8TraditionalOffset
- wc_CreatePKCS8Key
- wc_EncryptPKCS8Key
- wc_DecryptPKCS8Key

Return:

- The size of the encrypted key placed in out on success.
- LENGTH_ONLY_E if out is NULL, with required output buffer size in outSz.
- Other negative values on failure.

Example

```
byte* key;            // Traditional private key (DER formatted).
word32 keySz;         // Size of key.
byte* pkcs8Enc;       // Encrypted PKCS#8 key.
word32 pkcs8EncSz;    // Size of pkcs8Enc.
const char* password; // Password to use for encryption.
int passwordSz;       // Length of password (not including NULL terminator).
WC_RNG rng;

// The following produces an encrypted, PKCS#8 version of key in pkcs8Enc.
// The encryption uses password-based encryption scheme 2 (PBE2) from PKCS#5
// and the AES cipher in CBC mode with a 256-bit key. See RFC 8018 for more
// on PKCS#5.
ret = wc_CreateEncryptedPKCS8Key(key, keySz, pkcs8Enc, &pkcs8EncSz,
        password, passwordSz, PKCS5, PBES2, AES256CBCb, NULL, 0,
        WC_PKCS12_ITT_DEFAULT, &rng, NULL);
```

C.5.2.118 function wc_InitDecodedCert

```
void wc_InitDecodedCert(
    struct DecodedCert * cert,
    const byte * source,
    word32 inSz,
    void * heap
)
```

This function initializes the DecodedCert pointed to by the “cert” parameter. It saves the “source” pointer to a DER-encoded certificate of length “inSz.” This certificate can be parsed by a subsequent call to wc_ParseCert.

Parameters:

- **cert** Pointer to an allocated DecodedCert object.
- **source** Pointer to a DER-encoded certificate.
- **inSz** Length of the DER-encoded certificate in bytes.
- **heap** A pointer to the heap used for dynamic allocation. Can be NULL.

See:

- wc_ParseCert
- wc_FreeDecodedCert

Example

```
DecodedCert decodedCert; // Decoded certificate object.
byte* certBuf;           // DER-encoded certificate buffer.
word32 certBufSz;        // Size of certBuf in bytes.

wc_InitDecodedCert(&decodedCert, certBuf, certBufSz, NULL);
```

C.5.2.119 function wc_ParseCert

```
int wc_ParseCert(
    DecodedCert * cert,
    int type,
    int verify,
    WOLFSSL_CERT_MANAGER * cm
)
```

This function parses the DER-encoded certificate saved in the DecodedCert object and populates the fields of that object. The DecodedCert must have been initialized with a prior call to wc_InitDecodedCert. This function takes an optional pointer to a CertificateManager object, which is used to populate the certificate authority information of the DecodedCert, if the CA is found in the CertificateManager.

Parameters:

- **cert** Pointer to an initialized DecodedCert object.
- **type** Type of certificate. See the CertType enum in asn_public.h.
- **verify** Flag that, if set, indicates the user wants to verify the validity of the certificate.
- **cm** An optional pointer to a CertificateManager. Can be NULL.

See:

- wc_InitDecodedCert
- wc_FreeDecodedCert

Return:

- 0 on success.
- Other negative values on failure.

Example

```
int ret;
DecodedCert decodedCert; // Decoded certificate object.
byte* certBuf;           // DER-encoded certificate buffer.
word32 certBufSz;        // Size of certBuf in bytes.

wc_InitDecodedCert(&decodedCert, certBuf, certBufSz, NULL);
ret = wc_ParseCert(&decodedCert, CERT_TYPE, NO_VERIFY, NULL);
if (ret != 0) {
```

```
    fprintf(stderr, "wc_ParseCert failed.\n");
}
```

C.5.2.120 function wc_FreeDecodedCert

```
void wc_FreeDecodedCert(
    struct DecodedCert * cert
)
```

This function frees a DecodedCert that was previously initialized with wc_InitDecodedCert.

Parameters:

- **cert** Pointer to an initialized DecodedCert object.

See:

- wc_InitDecodedCert
- wc_ParseCert

Example

```
int ret;
DecodedCert decodedCert; // Decoded certificate object.
byte* certBuf;           // DER-encoded certificate buffer.
word32 certBufSz;        // Size of certBuf in bytes.

wc_InitDecodedCert(&decodedCert, certBuf, certBufSz, NULL);
ret = wc_ParseCert(&decodedCert, CERT_TYPE, NO_VERIFY, NULL);
if (ret != 0) {
    fprintf(stderr, "wc_ParseCert failed.\n");
}
wc_FreeDecodedCert(&decodedCert);
```

C.5.2.121 function wc_SetTimeCb

```
int wc_SetTimeCb(
    wc_time_cb f
)
```

This function registers a time callback that will be used anytime wolfSSL needs to get the current time. The prototype of the callback should be the same as the “time” function from the C standard library.

Parameters:

- **f** function to register as the time callback.

See: wc_Time

Return: 0 Returned on success.

Example

```
int ret = 0;
// Time callback prototype
time_t my_time_cb(time_t* t);
// Register it
ret = wc_SetTimeCb(my_time_cb);
if (ret != 0) {
    // failed to set time callback
}
```

```
time_t my_time_cb(time_t* t)
{
    // custom time function
}
```

C.5.2.122 function wc_Time

```
time_t wc_Time(
    time_t * t
)
```

This function gets the current time. By default, it uses the XTIME macro, which varies between platforms. The user can use a function of their choosing instead via the wc_SetTimeCb function.

Parameters:

- **t** Optional time_t pointer to populate with current time.

See: wc_SetTimeCb

Return: Time Current time returned on success.

Example

```
time_t currentTime = 0;
currentTime = wc_Time(NULL);
wc_Time(&currentTime);
```

C.5.2.123 function wc_SetCustomExtension

```
int wc_SetCustomExtension(
    Cert * cert,
    int critical,
    const char * oid,
    const byte * der,
    word32 derSz
)
```

This function injects a custom extension in to an X.509 certificate. note: The content at the address pointed to by any of the parameters that are pointers must not be modified until the certificate is generated and you have the der output. This function does NOT copy the contents to another buffer.

Parameters:

- **cert** Pointer to an initialized DecodedCert object.
- **critical** If 0, the extension will not be marked critical, otherwise it will be marked critical.
- **oid** Dot separated oid as a string. For example “1.2.840.10045.3.1.7”
- **der** The der encoding of the content of the extension.
- **derSz** The size in bytes of the der encoding.

See:

- wc_InitCert
- wc_SetUnknownExtCallback

Return:

- 0 Returned on success.
- Other negative values on failure.

Example

```
int ret = 0;
Cert newCert;
wc_InitCert(&newCert);

// Code to setup subject, public key, issuer, and other things goes here.

ret = wc_SetCustomExtension(&newCert, 1, "1.2.3.4.5",
          (const byte *)"This is a critical extension", 28);
if (ret < 0) {
    // Failed to set the extension.
}

ret = wc_SetCustomExtension(&newCert, 0, "1.2.3.4.6",
          (const byte *)"This is NOT a critical extension", 32)
if (ret < 0) {
    // Failed to set the extension.
}

// Code to sign the certificate and then write it out goes here.
```

C.5.2.124 function wc_SetUnknownExtCallback

```
int wc_SetUnknownExtCallback(
    DecodedCert * cert,
    wc_UnknownExtCallback cb
)
```

This function registers a callback that will be used anytime wolfSSL encounters an unknown X.509 extension in a certificate while parsing a certificate. The prototype of the callback should be:

Parameters:

- **cert** the DecodedCert struct that is to be associated with this callback.
- **cb** function to register as the time callback.

See:

- ParseCert
- wc_SetCustomExtension

Return:

- 0 Returned on success.
- Other negative values on failure.

Example

```
int ret = 0;
// Unknown extension callback prototype
int myUnknownExtCallback(const word16* oid, word32 oidSz, int crit,
                         const unsigned char* der, word32 derSz);

// Register it
ret = wc_SetUnknownExtCallback(cert, myUnknownExtCallback);
if (ret != 0) {
    // failed to set the callback
}

// oid: Array of integers that are the dot separated values in an oid.
```

```
// oidSz: Number of values in oid.
// crit: Whether the extension was mark critical.
// der: The der encoding of the content of the extension.
// derSz: The size in bytes of the der encoding.
int myCustomExtCallback(const word16* oid, word32 oidSz, int crit,
                        const unsigned char* der, word32 derSz) {

    // Logic to parse extension goes here.

    // NOTE: by returning zero, we are accepting this extension and
    // informing wolfSSL that it is acceptable. If you find an extension
    // that you do not find acceptable, you should return an error. The
    // standard behavior upon encountering an unknown extension with the
    // critical flag set is to return ASN_CRIT_EXT_E. For the sake of
    // brevity, this example is always accepting every extension; you
    // should use different logic.
    return 0;
}
```

C.5.2.125 function wc_CheckCertSigPubKey

```
int wc_CheckCertSigPubKey(
    const byte * cert,
    word32 certSz,
    void * heap,
    const byte * pubKey,
    word32 pubKeySz,
    int pubKeyOID
)
```

This function verifies the signature in the der form of an X.509 certificate against a public key. The public key is expected to be the full subject public key info in der form.

Parameters:

- **cert** The der encoding of the X.509 certificate.
- **certSz** The size in bytes of cert.
- **heap** A pointer to the heap used for dynamic allocation. Can be NULL.
- **pubKey** The der encoding of the public key.
- **pubKeySz** The size in bytes of pubKey.
- **pubKeyOID** OID identifying the algorithm of the public key. (ie: ECDSAk, DSAk or RSAk)

Return:

- 0 Returned on success.
- Other negative values on failure.

C.5.2.126 function wc_Asn1PrintOptions_Init

```
int wc_Asn1PrintOptions_Init(
    Asn1PrintOptions * opts
)
```

This function initializes the ASN.1 print options.

Parameters:

- **opts** The ASN.1 options for printing.

See:

- wc_Asn1PrintOptions_Set
- wc_Asn1_PrintAll

Return:

- 0 on success.
- BAD_FUNC_ARG when asn1 is NULL.

Example

```
Asn1PrintOptions opt;

// Initialize ASN.1 print options before use.
wc_Asn1PrintOptions_Init(&opt);
```

C.5.2.127 function wc_Asn1PrintOptions_Set

```
int wc_Asn1PrintOptions_Set(
    Asn1PrintOptions * opts,
    enum Asn1PrintOpt opt,
    word32 val
)
```

This function sets a print option into an ASN.1 print options object.

Parameters:

- **opts** The ASN.1 options for printing.
- **opt** An option to set value for.
- **val** The value to set.

See:

- wc_Asn1PrintOptions_Init
- wc_Asn1_PrintAll

Return:

- 0 on success.
- BAD_FUNC_ARG when asn1 is NULL.
- BAD_FUNC_ARG when val is out of range for option.

Example

```
Asn1PrintOptions opt;

// Initialize ASN.1 print options before use.
wc_Asn1PrintOptions_Init(&opt);
// Set the number of indents when printing tag name to be 1.
wc_Asn1PrintOptions_Set(&opt, ASN1_PRINT_OPT_INDENT, 1);
```

C.5.2.128 function wc_Asn1_Init

```
int wc_Asn1_Init(
    Asn1 * asn1
)
```

This function initializes an ASN.1 parsing object.

Parameters:

- **asn1** ASN.1 parse object.

See:

- wc_Asn1_SetFile
- wc_Asn1_PrintAll

Return:

- 0 on success.
- BAD_FUNC_ARG when asn1 is NULL.

Example

```
Asn1 asn1;

// Initialize ASN.1 parse object before use.
wc_Asn1_Init(&asn1);
```

C.5.2.129 function wc_Asn1_SetFile

```
int wc_Asn1_SetFile(
    Asn1 * asn1,
    XFILE file
)
```

This function sets the file to use when printing into an ASN.1 parsing object.

Parameters:

- **asn1** The ASN.1 parse object.
- **file** File to print to.

See:

- wc_Asn1_Init
- wc_Asn1_PrintAll

Return:

- 0 on success.
- BAD_FUNC_ARG when asn1 is NULL.
- BAD_FUNC_ARG when file is XBADFILE.

Example

```
Asn1 asn1;

// Initialize ASN.1 parse object before use.
wc_Asn1_Init(&asn1);
// Set standard out to be the file descriptor to write to.
wc_Asn1_SetFile(&asn1, stdout);
```

C.5.2.130 function wc_Asn1_PrintAll

```
int wc_Asn1_PrintAll(
    Asn1 * asn1,
    Asn1PrintOptions * opts,
    unsigned char * data,
    word32 len
)
```

Print all ASN.1 items.

Parameters:

- **asn1** The ASN.1 parse object.
- **opts** The ASN.1 print options.
- **data** Buffer containing BER/DER data to print.
- **len** Length of data to print in bytes.

See:

- wc_Asn1_Init
- wc_Asn1_SetFile

Return:

- 0 on success.
- BAD_FUNC_ARG when asn1 or opts is NULL.
- ASN_LEN_E when ASN.1 item's length too long.
- ASN_DEPTH_E when end offset invalid.
- ASN_PARSE_E when not all of an ASN.1 item parsed.

```
Asn1PrintOptions opts;
Asn1 asn1;
unsigned char data[] = { Initialize with DER/BER data };
word32 len = sizeof(data);

// Initialize ASN.1 print options before use.
wc_Asn1PrintOptions_Init(&opt);
// Set the number of indents when printing tag name to be 1.
wc_Asn1PrintOptions_Set(&opt, ASN1_PRINT_OPT_INDENT, 1);

// Initialize ASN.1 parse object before use.
wc_Asn1_Init(&asn1);
// Set standard out to be the file descriptor to write to.
wc_Asn1_SetFile(&asn1, stdout);
// Print all ASN.1 items in buffer with the specified print options.
wc_Asn1_PrintAll(&asn1, &opts, data, len);
```

C.5.2.131 function wc_Asn1_SetOidToNameCb

```
int wc_Asn1_SetOidToNameCb(
    Asn1 * asn1,
    Asn1OidToNameCb nameCb
)
```

Sets OID to name callback for ASN.1 parsing.

Parameters:

- **asn1** ASN.1 structure
- **nameCb** Callback function to convert OID to name

See: wc_Asn1_PrintAll

Return:

- 0 on success
- negative on error

Example

```
Asn1 asn1;
int ret = wc_Asn1_SetOidToNameCb(&asn1, myOidToNameCb);
```

C.5.2.132 function wc_GetDecodedCertSubjectRaw

```
int wc_GetDecodedCertSubjectRaw(
    const struct DecodedCert * cert,
    const byte ** subjectRaw,
    int * subjectRawSz
)
```

Retrieves the raw DER-encoded subject Name content from a parsed DecodedCert.

Parameters:

- **cert** Pointer to the DecodedCert (must have been parsed).
- **subjectRaw** Output pointer that receives the address of the raw DER subject content.
- **subjectRawSz** Output pointer that receives the size in bytes of the raw subject content.

See:

- wc_InitDecodedCert
- wc_ParseCert
- wc_FreeDecodedCert
- wc_GetDecodedCertSubject
- wolfSSL_UseCertificateAuthority

Return:

- 0 on success.
- BAD_FUNC_ARG if any argument is NULL.
- ASN_PARSE_E if the subject was not populated during parsing.
- NOT_COMPILED_IN if the required build options are not enabled.

The returned pointer and size reference the inner content of the subject Name SEQUENCE (i.e. the bytes after the SEQUENCE tag and length). The pointer aliases the DER buffer supplied to wc_InitDecodedCert(), which the DecodedCert does not own, and must not be freed by the caller. The data remains valid only while that buffer is alive and unmodified.

This function is intended for use with wolfSSL_UseCertificateAuthority(), which expects the subject content without the outer SEQUENCE header.

Requires IGNORE_NAME_CONSTRAINTS to be undefined or WOLFSSL_CERT_EXT to be defined.

Example

```
DecodedCert decoded;
const byte* subject = NULL;
int subjectSz = 0;

wc_InitDecodedCert(&decoded, certDer, certDerSz, NULL);
if (wc_ParseCert(&decoded, CERT_TYPE, NO_VERIFY, NULL) == 0) {
    if (wc_GetDecodedCertSubjectRaw(&decoded, &subject,
            &subjectSz) == 0) {
        // subject and subjectSz now reference the raw DER content
    }
}
wc_FreeDecodedCert(&decoded);
```

C.5.3 Source code

```
int wc_InitCert(Cert* cert);

Cert* wc_CertNew(void* heap);

int wc_InitCert_ex(Cert* cert, void* heap, int devId);

void  wc_CertFree(Cert* cert);

int  wc_MakeCert(Cert* cert, byte* derBuffer, word32 derSz, RsaKey* rsaKey,
                             ecc_key* eccKey, WC_RNG* rng);

int wc_MakeCert_ex(Cert* cert, byte* derBuffer, word32 derSz,
                   int keyType, void* key, WC_RNG* rng);

int wc_MakeCertReq_ex(Cert* cert, byte* derBuffer, word32 derSz,
                      int keyType, void* key);

int wc_SignCert_ex(int requestSz, int sType, byte* buf, word32 buffSz,
                   int keyType, void* key, WC_RNG* rng);

int wc_MakeSigWithBitStr(byte *sig, int sigSz, int sType, byte* buf,
                         word32 bufSz, int keyType, void* key,
                         WC_RNG* rng);

int wc_GetCertDates(Cert* cert, struct tm* before, struct tm* after);

int wc_GetDateInfo(const byte* certDate, int certDateSz,
                   const byte** date, byte* format, int* length);

int wc_GetDateAsCalendarTime(const byte* date, int length,
                             byte format, struct tm* timearg);

int  wc_MakeCertReq(Cert* cert, byte* derBuffer, word32 derSz,
                                 RsaKey* rsaKey, ecc_key* eccKey);

int  wc_SignCert(int requestSz, int sigType, byte* derBuffer,
                 word32 derSz, RsaKey* rsaKey, ecc_key* eccKey, WC_RNG* rng);

int  wc_MakeSelfCert(Cert* cert, byte* derBuffer, word32 derSz, RsaKey* key,
                             WC_RNG* rng);

int  wc_SetIssuer(Cert* cert, const char* issuerFile);

int  wc_SetSubject(Cert* cert, const char* subjectFile);


int  wc_SetSubjectRaw(Cert* cert, const byte* der, int derSz);

int  wc_GetSubjectRaw(byte **subjectRaw, Cert *cert);

int  wc_SetAltNames(Cert* cert, const char* file);
```

```
int  wc_SetIssuerBuffer(Cert* cert, const byte* der, int derSz);

int  wc_SetIssuerRaw(Cert* cert, const byte* der, int derSz);

int  wc_SetSubjectBuffer(Cert* cert, const byte* der, int derSz);

int  wc_SetAltNamesBuffer(Cert* cert, const byte* der, int derSz);

int wc_SetDNSEntry(void* heap, const char* str, int strLen, int type,
                   DNS_entry** entries);

int wc_FlattenAltNames(byte* output, word32 outputSz, const DNS_entry* names);

int wc_SetAltNamesFromList(Cert* cert, const DNS_entry* names);

int  wc_SetDatesBuffer(Cert* cert, const byte* der, int derSz);

int wc_SetAuthKeyIdFromPublicKey(Cert *cert, RsaKey *rsakey,
                                             ecc_key *eckey);

int wc_SetAuthKeyIdFromPublicKey_ex(Cert *cert, int keyType,
                                    void* key);

int wc_SetAuthKeyIdFromCert(Cert *cert, const byte *der, int derSz);

int wc_SetAuthKeyId(Cert *cert, const char* file);

int wc_SetSubjectKeyIdFromPublicKey(Cert *cert, RsaKey *rsakey,
                                                ecc_key *eckey);

int wc_SetSubjectKeyIdFromPublicKey_ex(Cert *cert, int keyType,
                                       void* key);

int wc_SetSubjectKeyId(Cert *cert, const char* file);

int wc_SetKeyUsage(Cert *cert, const char *value);

int wc_SetExtKeyUsage(Cert *cert, const char *value);

int wc_SetExtKeyUsageOID(Cert *cert, const char *oid, word32 sz,
                         byte idx, void* heap);

int wc_PemPubKeyToDer(const char* fileName,
                                       unsigned char* derBuf, int derSz);

int wc_PemPubKeyToDer_ex(const char* fileName, DerBuffer** der);

int wc_PubKeyPemToDer(const unsigned char* pem, int pemSz,
                                        unsigned char* buff, int buffSz);

int wc_PemGetHeaderFooter(int type, const char** header,
                          const char** footer);
```

```
int wc_AllocDer(DerBuffer** pDer, word32 length, int type,
                void* heap);

void wc_FreeDer(DerBuffer** pDer);

int wc_PemToDer(const unsigned char* buff, long longSz, int type,
                DerBuffer** pDer, void* heap, EncryptedInfo* info,
                int* keyFormat);

int wc_PemCertToDer(const char* fileName, unsigned char* derBuf, int derSz);

int wc_DerToPem(const byte* der, word32 derSz, byte* output,
                                word32 outSz, int type);

int wc_DerToPemEx(const byte* der, word32 derSz, byte* output,
                                word32 outSz, byte *cipher_info, int type);

int wc_KeyPemToDer(const unsigned char* pem, int pemSz,
                                   unsigned char* buff, int buffSz, const char*
                                     ↪  pass);

int wc_CertPemToDer(const unsigned char* pem, int pemSz,
                    unsigned char* buff, int buffSz, int type);

int wc_PemCertToDer_ex(const char* fileName, DerBuffer** der);

word32 wc_PkcsPad(byte* buf, word32 sz, word32 blockSz);

int wc_RsaPublicKeyDecode_ex(const byte* input, word32* inOutIdx,
                             word32 inSz, const byte** n, word32* nSz,
                             const byte** e, word32* eSz);

int wc_RsaPublicKeyDerSize(RsaKey* key, int with_header);

int wc_RsaPrivateKeyValidate(const byte* input, word32* inOutIdx,
                             int* keySz, word32 inSz);

int wc_GetPubKeyDerFromCert(struct DecodedCert* cert,
                                       byte* derKey, word32* derKeySz);

int wc_DsaParamsDecode(const byte* input, word32* inOutIdx,
                       DsaKey* key, word32 inSz);

int wc_DsaKeyToParamsDer(DsaKey* key, byte* output, word32 outLen);

int wc_DsaKeyToParamsDer_ex(DsaKey* key, byte* output,
                            word32* outLen);

int wc_DhParamsToDer(DhKey* key, byte* out, word32* outSz);

int wc_DhPubKeyToDer(DhKey* key, byte* out, word32* outSz);

int wc_DhPrivKeyToDer(DhKey* key, byte* out, word32* outSz);
```

```
int wc_EccPrivateKeyDecode(const byte* input, word32* inOutIdx,
                                          ecc_key* key, word32 inSz);

int wc_EccPrivateKeyToDer(ecc_key* key, byte* output,
                          word32 outLen);

int wc_EccKeyDerSize(ecc_key* key, int pub);

int wc_EccPrivateKeyToPKCS8(ecc_key* key, byte* output,
                            word32* outLen);

int wc_EccKeyToPKCS8(ecc_key* key, byte* output,
                     word32* outLen);

int wc_EccPublicKeyDerSize(ecc_key* key, int with_AlgCurve);

int wc_EccKeyToDer(ecc_key* key, byte* output, word32 outLen);

int wc_EccPublicKeyDecode(const byte* input, word32* inOutIdx,
                          ecc_key* key, word32 inSz);

int wc_EccPublicKeyToDer(ecc_key* key, byte* output,
                                       word32 outLen, int with_AlgCurve);

int wc_EccPublicKeyToDer_ex(ecc_key* key, byte* output,
                                    word32 outLen, int with_AlgCurve, int comp);


int wc_Curve25519PrivateKeyDecode(const byte* input, word32* inOutIdx,
                                  curve25519_key* key, word32 inSz);

int wc_Curve25519PublicKeyDecode(const byte* input, word32* inOutIdx,
                                 curve25519_key* key, word32 inSz);

int wc_Curve25519KeyDecode(const byte* input, word32* inOutIdx,
                           curve25519_key* key, word32 inSz);

int wc_Curve25519PrivateKeyToDer(curve25519_key* key, byte* output,
                                 word32 outLen);

int wc_Curve25519PublicKeyToDer(curve25519_key* key, byte* output, word32
 ↪  outLen,
                                int withAlg);

int wc_Curve25519KeyToDer(curve25519_key* key, byte* output, word32 outLen,
                          int withAlg);

int wc_Ed25519PrivateKeyDecode(const byte* input, word32* inOutIdx,
                               ed25519_key* key, word32 inSz);

int wc_Ed25519PrivateKeyDecode_ex(const byte* input, word32* inOutIdx,
                                  ed25519_key* key, word32 inSz, int trusted);

int wc_Ed25519PublicKeyDecode(const byte* input, word32* inOutIdx,
```

```
                                ed25519_key* key, word32 inSz);

int wc_Ed25519KeyToDer(const ed25519_key* key, byte* output,
                       word32 outLen);

int wc_Ed25519PrivateKeyToDer(const ed25519_key* key, byte* output,
                              word32 outLen);

int wc_Ed25519PublicKeyToDer(const ed25519_key* key, byte* output,
        word32 outLen, int withAlg);

int wc_Ed448PrivateKeyDecode(const byte* input, word32* inOutIdx,
                             ed448_key* key, word32 inSz);

int wc_Ed448PrivateKeyDecode_ex(const byte* input, word32* inOutIdx,
                                ed448_key* key, word32 inSz, int trusted);

int wc_Ed448PublicKeyDecode(const byte* input, word32* inOutIdx,
                            ed448_key* key, word32 inSz);

int wc_Ed448KeyToDer(const ed448_key* key, byte* output, word32 outLen);

int wc_Ed448PrivateKeyToDer(const ed448_key* key, byte* output,
                            word32 outLen);

int wc_Ed448PublicKeyToDer(const ed448_key* key, byte* output,
                           word32 outLen, int withAlg);

int wc_Curve448PrivateKeyDecode(const byte* input, word32* inOutIdx,
                                curve448_key* key, word32 inSz);

int wc_Curve448PublicKeyDecode(const byte* input, word32* inOutIdx,
                               curve448_key* key, word32 inSz);

int wc_Curve448PrivateKeyToDer(curve448_key* key, byte* output,
                               word32 outLen);

int wc_Curve448PublicKeyToDer(curve448_key* key, byte* output,
                              word32 outLen);

word32 wc_EncodeSignature(byte* out, const byte* digest,
                                     word32 digSz, int hashOID);

int wc_GetCTC_HashOID(int type);

void wc_SetCert_Free(Cert* cert);

int wc_GetPkcs8TraditionalOffset(byte* input,
                                              word32* inOutIdx, word32 sz);

int wc_CreatePKCS8Key(byte* out, word32* outSz,
        byte* key, word32 keySz, int algoID, const byte* curveOID,
        word32 oidSz);
```

```
int wc_EncryptPKCS8Key(byte* key, word32 keySz, byte* out,
        word32* outSz, const char* password, int passwordSz, int vPKCS,
        int pbeOid, int encAlgId, byte* salt, word32 saltSz, int itt,
        WC_RNG* rng, void* heap);

int wc_EncryptPKCS8Key_ex(byte* key, word32 keySz, byte* out,
                          word32* outSz, const char* password,
                          int passwordSz, int vPKCS, int pbeOid,
                          int encAlgId, byte* salt, word32 saltSz,
                          int itt, WC_RNG* rng, void* heap,
                          int devId);

int wc_GetTime(void* timePtr, word32 timeSize);

int wc_EncryptedInfoGet(EncryptedInfo* info,
                        const char* cipherName);

int wc_ParseCertPIV(wc_CertPIV* cert, const byte* buf,
                    word32 totalSz);

int wc_GetSubjectPubKeyInfoDerFromCert(const byte* certDer,
                                       word32 certDerSz,
                                       byte* pubKeyDer,
                                       word32* pubKeyDerSz);

int wc_GetDecodedCertSubject(const struct DecodedCert* cert,
                             char* buf, word32* bufSz);

int wc_GetDecodedCertIssuer(const struct DecodedCert* cert,
                            char* buf, word32* bufSz);

int wc_GetDecodedCertSerial(const struct DecodedCert* cert,
                            byte* buf, word32* bufSz);

int wc_GetUUIDFromCert(struct DecodedCert* cert,
                       byte* uuid, int* uuidSz);

int wc_GetFASCNFromCert(struct DecodedCert* cert,
                        byte* fascn, int* fascnSz);

int wc_GeneratePreTBS(struct DecodedCert* cert, byte *der,
                      int derSz);

void wc_InitDecodedAcert(struct DecodedAcert* acert,
                         void* heap);

void wc_FreeDecodedAcert(struct DecodedAcert * acert);

int wc_ParseX509Acert(struct DecodedAcert* acert, int verify);

int wc_VerifyX509Acert(const byte* acert, word32 acertSz,
                       const byte* issuerCert,
                       word32 issuerCertSz, void* cm);
```

```
int wc_DecryptPKCS8Key(byte* input, word32 sz, const char* password,
        int passwordSz);

int wc_CreateEncryptedPKCS8Key(byte* key, word32 keySz, byte* out,
        word32* outSz, const char* password, int passwordSz, int vPKCS,
        int pbeOid, int encAlgId, byte* salt, word32 saltSz, int itt,
        WC_RNG* rng, void* heap);

void wc_InitDecodedCert(struct DecodedCert* cert,
    const byte* source, word32 inSz, void* heap);

int wc_ParseCert(DecodedCert* cert, int type, int verify,
    WOLFSSL_CERT_MANAGER* cm);

void wc_FreeDecodedCert(struct DecodedCert* cert);

int wc_SetTimeCb(wc_time_cb f);

time_t wc_Time(time_t* t);

int wc_SetCustomExtension(Cert *cert, int critical, const char *oid,
                                      const byte *der, word32 derSz);

int wc_SetUnknownExtCallback(DecodedCert* cert,
                                           wc_UnknownExtCallback cb);
int wc_CheckCertSigPubKey(const byte* cert, word32 certSz,
                                      void* heap, const byte* pubKey,
                                      word32 pubKeySz, int pubKeyOID);

int wc_Asn1PrintOptions_Init(Asn1PrintOptions* opts);

int wc_Asn1PrintOptions_Set(Asn1PrintOptions* opts, enum Asn1PrintOpt opt,
    word32 val);

int wc_Asn1_Init(Asn1* asn1);

int wc_Asn1_SetFile(Asn1* asn1, XFILE file);

int wc_Asn1_PrintAll(Asn1* asn1, Asn1PrintOptions* opts, unsigned char* data,
    word32 len);

int wc_Asn1_SetOidToNameCb(Asn1* asn1, Asn1OidToNameCb nameCb);

int wc_GetDecodedCertSubjectRaw(const struct DecodedCert* cert,
                                const byte** subjectRaw,
                                int* subjectRawSz);
```

C.6 blake2.h

C.6.1 Functions

	Name
int	**wc_InitBlake2b**(Blake2b * b2b, word32 digestSz)This function initializes a Blake2b structure for use with the Blake2 hash function.
int	**wc_Blake2bUpdate**(Blake2b * b2b, const byte * data, word32 sz)This function updates the Blake2b hash with the given input data. This function should be called after wc_InitBlake2b, and repeated until one is ready for the final hash: wc_Blake2bFinal.
int	**wc_Blake2bFinal**(Blake2b * b2b, byte * final, word32 requestSz)This function computes the Blake2b hash of the previously supplied input data. The output hash will be of length requestSz, or, if requestSz==0, the digestSz of the b2b structure. This function should be called after wc_InitBlake2b and wc_Blake2bUpdate has been processed for each piece of input data desired.
int	**wc_Blake2bHmacInit**(Blake2b * b2b, const byte * key, size_t key_len)Initialize an HMAC-BLAKE2b message authentication code computation.
int	**wc_Blake2bHmacUpdate**(Blake2b * b2b, const byte * in, size_t in_len)Update an HMAC-BLAKE2b message authentication code computation with additional input data.
int	**wc_Blake2bHmacFinal**(Blake2b * b2b, const byte * key, size_t key_len, byte * out, size_t out_len)Finalize an HMAC-BLAKE2b message authentication code computation.
int	**wc_Blake2bHmac**(const byte * in, size_t in_len, const byte * key, size_t key_len, byte * out, size_t out_len)Compute the HMAC-BLAKE2b message authentication code of the given input data using the given key.
int	**wc_InitBlake2s**(Blake2s * b2s, word32 digestSz)This function initializes a Blake2s structure for use with the Blake2 hash function.
int	**wc_Blake2sUpdate**(Blake2s * b2s, const byte * data, word32 sz)This function updates the Blake2s hash with the given input data. This function should be called after wc_InitBlake2s, and repeated until one is ready for the final hash: wc_Blake2sFinal.

	Name
int	**wc_Blake2sFinal**(Blake2s * b2s, byte * final, word32 requestSz)This function computes the Blake2s hash of the previously supplied input data. The output hash will be of length requestSz, or, if requestSz==0, the digestSz of the b2s structure. This function should be called after wc_InitBlake2s and wc_Blake2sUpdate has been processed for each piece of input data desired.
int	**wc_Blake2sHmacInit**(Blake2s * b2s, const byte * key, size_t key_len)Initialize an HMAC-BLAKE2s message authentication code computation.
int	**wc_Blake2sHmacUpdate**(Blake2s * b2s, const byte * in, size_t in_len)Update an HMAC-BLAKE2s message authentication code computation with additional input data.
int	**wc_Blake2sHmacFinal**(Blake2s * b2s, const byte * key, size_t key_len, byte * out, size_t out_len)Finalize an HMAC-BLAKE2s message authentication code computation.
int	**wc_Blake2sHmac**(const byte * in, size_t in_len, const byte * key, size_t key_len, byte * out, size_t out_len)This function computes the HMAC-BLAKE2s message authentication code of the given input data using the given key.

C.6.2 Functions Documentation

C.6.2.1 function wc_InitBlake2b

```
int wc_InitBlake2b(
    Blake2b * b2b,
    word32 digestSz
)
```

This function initializes a Blake2b structure for use with the Blake2 hash function.

Parameters:

- **b2b** pointer to the Blake2b structure to initialize
- **digestSz** length of the blake 2 digest to implement

See: wc_Blake2bUpdate

Return: 0 Returned upon successfully initializing the Blake2b structure and setting the digest size.

Example

```
Blake2b b2b;
// initialize Blake2b structure with 64 byte digest
wc_InitBlake2b(&b2b, WC_BLAKE2B_DIGEST_SIZE);
```

C.6.2.2 function wc_Blake2bUpdate

```
int wc_Blake2bUpdate(
    Blake2b * b2b,
    const byte * data,
    word32 sz
)
```

This function updates the Blake2b hash with the given input data. This function should be called after wc_InitBlake2b, and repeated until one is ready for the final hash: wc_Blake2bFinal.

Parameters:

- **b2b** pointer to the Blake2b structure to update
- **data** pointer to a buffer containing the data to append
- **sz** length of the input data to append

See:

- wc_InitBlake2b
- wc_Blake2bFinal

Return:

- 0 Returned upon successfully update the Blake2b structure with the given data
- -1 Returned if there is a failure while compressing the input data

Example

```
int ret;
Blake2b b2b;
// initialize Blake2b structure with 64 byte digest
wc_InitBlake2b(&b2b, WC_BLAKE2B_DIGEST_SIZE);

byte plain[] = { // initialize input };

ret = wc_Blake2bUpdate(&b2b, plain, sizeof(plain));
if (ret != 0) {
    // error updating blake2b
}
```

C.6.2.3 function wc_Blake2bFinal

```
int wc_Blake2bFinal(
    Blake2b * b2b,
    byte * final,
    word32 requestSz
)
```

This function computes the Blake2b hash of the previously supplied input data. The output hash will be of length requestSz, or, if requestSz==0, the digestSz of the b2b structure. This function should be called after wc_InitBlake2b and wc_Blake2bUpdate has been processed for each piece of input data desired.

Parameters:

- **b2b** pointer to the Blake2b structure to update
- **final** pointer to a buffer in which to store the blake2b hash. Should be of length requestSz
- **requestSz** length of the digest to compute. When this is zero, b2b->digestSz will be used instead

See:

- wc_InitBlake2b

- wc_Blake2bUpdate

Return:

- 0 Returned upon successfully computing the Blake2b hash
- -1 Returned if there is a failure while parsing the Blake2b hash

Example

```
int ret;
Blake2b b2b;
byte hash[WC_BLAKE2B_DIGEST_SIZE];
// initialize Blake2b structure with 64 byte digest
wc_InitBlake2b(&b2b, WC_BLAKE2B_DIGEST_SIZE);
... // call wc_Blake2bUpdate to add data to hash

ret = wc_Blake2bFinal(&b2b, hash, WC_BLAKE2B_DIGEST_SIZE);
if (ret != 0) {
    // error generating blake2b hash
}
```

C.6.2.4 function wc_Blake2bHmacInit

```
int wc_Blake2bHmacInit(
    Blake2b * b2b,
    const byte * key,
    size_t key_len
)
```

Initialize an HMAC-BLAKE2b message authentication code computation.

Parameters:

- **b2b** Blake2b structure to be used for the MAC computation.
- **key** pointer to the key
- **key_len** length of the key

Return: 0 Returned upon successfully initializing the HMAC-BLAKE2b MAC computation.

Example

```
Blake2b b2b;
int ret;
byte key[] = {4, 5, 6};
ret = wc_Blake2bHmacInit(&b2b, key);
if (ret != 0) {
    // error generating HMAC-BLAKE2b
}
```

C.6.2.5 function wc_Blake2bHmacUpdate

```
int wc_Blake2bHmacUpdate(
    Blake2b * b2b,
    const byte * in,
    size_t in_len
)
```

Update an HMAC-BLAKE2b message authentication code computation with additional input data.

Parameters:

- **b2b** Blake2b structure to be used for the MAC computation.
- **in** pointer to the input data
- **in_len** length of the input data

Return: 0 Returned upon successfully updating the HMAC-BLAKE2b MAC computation.

Example

```
Blake2b b2b;
int ret;
byte key[] = {4, 5, 6};
byte data[] = {1, 2, 3};
ret = wc_Blake2bHmacInit(&b2b, key, sizeof(key));
ret = wc_Blake2bHmacUpdate(&b2b, data, sizeof(data));
```

C.6.2.6 function wc_Blake2bHmacFinal

```
int wc_Blake2bHmacFinal(
    Blake2b * b2b,
    const byte * key,
    size_t key_len,
    byte * out,
    size_t out_len
)
```

Finalize an HMAC-BLAKE2b message authentication code computation.

Parameters:

- **b2b** Blake2b structure to be used for the MAC computation.
- **key** pointer to the key
- **key_len** length of the key
- **out** output buffer to store computed MAC
- **out_len** length of output buffer

Return: 0 Returned upon successfully finalizing the HMAC-BLAKE2b MAC computation.

Example

```
Blake2b b2b;
int ret;
byte key[] = {4, 5, 6};
byte data[] = {1, 2, 3};
byte mac[WC_BLAKE2B_DIGEST_SIZE];
ret = wc_Blake2bHmacInit(&b2b, key, sizeof(key));
ret = wc_Blake2bHmacUpdate(&b2b, data, sizeof(data));
ret = wc_Blake2bHmacFinalize(&b2b, key, sizeof(key), mac, sizezof(mac));
```

C.6.2.7 function wc_Blake2bHmac

```
int wc_Blake2bHmac(
    const byte * in,
    size_t in_len,
    const byte * key,
    size_t key_len,
    byte * out,
    size_t out_len
)
```

Compute the HMAC-BLAKE2b message authentication code of the given input data using the given key.

Parameters:

- **in** pointer to the input data
- **in_len** length of the input data
- **key** pointer to the key
- **key_len** length of the key
- **out** output buffer to store computed MAC
- **out_len** length of output buffer

Return: 0 Returned upon successfully computing the HMAC-BLAKE2b MAC.

Example

```
int ret;
byte mac[WC_BLAKE2B_DIGEST_SIZE];
byte data[] = {1, 2, 3};
byte key[] = {4, 5, 6};
ret = wc_Blake2bHmac(data, sizeof(data), key, sizeof(key), mac, sizeof(mac));
if (ret != 0) {
    // error generating HMAC-BLAKE2b
}
```

C.6.2.8 function wc_InitBlake2s

```
int wc_InitBlake2s(
    Blake2s * b2s,
    word32 digestSz
)
```

This function initializes a Blake2s structure for use with the Blake2 hash function.

Parameters:

- **b2s** pointer to the Blake2s structure to initialize
- **digestSz** length of the blake 2 digest to implement

See: wc_Blake2sUpdate

Return: 0 Returned upon successfully initializing the Blake2s structure and setting the digest size.

Example

```
Blake2s b2s;
// initialize Blake2s structure with 32 byte digest
wc_InitBlake2s(&b2s, WC_BLAKE2S_DIGEST_SIZE);
```

C.6.2.9 function wc_Blake2sUpdate

```
int wc_Blake2sUpdate(
    Blake2s * b2s,
    const byte * data,
    word32 sz
)
```

This function updates the Blake2s hash with the given input data. This function should be called after wc_InitBlake2s, and repeated until one is ready for the final hash: wc_Blake2sFinal.

Parameters:

- **b2s** pointer to the Blake2s structure to update
- **data** pointer to a buffer containing the data to append
- **sz** length of the input data to append

See:

- wc_InitBlake2s
- wc_Blake2sFinal

Return:

- 0 Returned upon successfully update the Blake2s structure with the given data
- -1 Returned if there is a failure while compressing the input data

Example

```
int ret;
Blake2s b2s;
// initialize Blake2s structure with 32 byte digest
wc_InitBlake2s(&b2s, WC_BLAKE2S_DIGEST_SIZE);

byte plain[] = { // initialize input };

ret = wc_Blake2sUpdate(&b2s, plain, sizeof(plain));
if (ret != 0) {
    // error updating blake2s
}
```

C.6.2.10 function wc_Blake2sFinal

```
int wc_Blake2sFinal(
    Blake2s * b2s,
    byte * final,
    word32 requestSz
)
```

This function computes the Blake2s hash of the previously supplied input data. The output hash will be of length requestSz, or, if requestSz==0, the digestSz of the b2s structure. This function should be called after wc_InitBlake2s and wc_Blake2sUpdate has been processed for each piece of input data desired.

Parameters:

- **b2s** pointer to the Blake2s structure to update
- **final** pointer to a buffer in which to store the blake2s hash. Should be of length requestSz
- **requestSz** length of the digest to compute. When this is zero, b2s->digestSz will be used instead

See:

- wc_InitBlake2s
- wc_Blake2sUpdate

Return:

- 0 Returned upon successfully computing the Blake2s hash
- -1 Returned if there is a failure while parsing the Blake2s hash

Example

```
int ret;
Blake2s b2s;
byte hash[WC_BLAKE2S_DIGEST_SIZE];
```

```
// initialize Blake2s structure with 32 byte digest
wc_InitBlake2s(&b2s, WC_BLAKE2S_DIGEST_SIZE);
... // call wc_Blake2sUpdate to add data to hash

ret = wc_Blake2sFinal(&b2s, hash, WC_BLAKE2S_DIGEST_SIZE);
if (ret != 0) {
    // error generating blake2s hash
}
```

C.6.2.11 function wc_Blake2sHmacInit

```
int wc_Blake2sHmacInit(
    Blake2s * b2s,
    const byte * key,
    size_t key_len
)
```

Initialize an HMAC-BLAKE2s message authentication code computation.

Parameters:

- **b2s** Blake2s structure to be used for the MAC computation.
- **key** pointer to the key
- **key_len** length of the key

Return: 0 Returned upon successfully initializing the HMAC-BLAKE2s MAC computation.

Example

```
Blake2s b2s;
int ret;
byte key[] = {4, 5, 6};
ret = wc_Blake2sHmacInit(&b2s, key);
if (ret != 0) {
    // error generating HMAC-BLAKE2s
}
```

C.6.2.12 function wc_Blake2sHmacUpdate

```
int wc_Blake2sHmacUpdate(
    Blake2s * b2s,
    const byte * in,
    size_t in_len
)
```

Update an HMAC-BLAKE2s message authentication code computation with additional input data.

Parameters:

- **b2s** Blake2s structure to be used for the MAC computation.
- **in** pointer to the input data
- **in_len** length of the input data

Return: 0 Returned upon successfully updating the HMAC-BLAKE2s MAC computation.

Example

```
Blake2s b2s;
int ret;
byte key[] = {4, 5, 6};
```

```
byte data[] = {1, 2, 3};
ret = wc_Blake2sHmacInit(&b2s, key, sizeof(key));
ret = wc_Blake2sHmacUpdate(&b2s, data, sizeof(data));
```

C.6.2.13 function wc_Blake2sHmacFinal

```
int wc_Blake2sHmacFinal(
    Blake2s * b2s,
    const byte * key,
    size_t key_len,
    byte * out,
    size_t out_len
)
```

Finalize an HMAC-BLAKE2s message authentication code computation.

Parameters:

- **b2s** Blake2s structure to be used for the MAC computation.
- **key** pointer to the key
- **key_len** length of the key
- **out** output buffer to store computed MAC
- **out_len** length of output buffer

Return: 0 Returned upon successfully finalizing the HMAC-BLAKE2s MAC computation.

Example

```
Blake2s b2s;
int ret;
byte key[] = {4, 5, 6};
byte data[] = {1, 2, 3};
byte mac[WC_BLAKE2S_DIGEST_SIZE];
ret = wc_Blake2sHmacInit(&b2s, key, sizeof(key));
ret = wc_Blake2sHmacUpdate(&b2s, data, sizeof(data));
ret = wc_Blake2sHmacFinalize(&b2s, key, sizeof(key), mac, sizezof(mac));
```

C.6.2.14 function wc_Blake2sHmac

```
int wc_Blake2sHmac(
    const byte * in,
    size_t in_len,
    const byte * key,
    size_t key_len,
    byte * out,
    size_t out_len
)
```

This function computes the HMAC-BLAKE2s message authentication code of the given input data using the given key.

Parameters:

- **in** pointer to the input data
- **in_len** length of the input data
- **key** pointer to the key
- **key_len** length of the key
- **out** output buffer to store computed MAC
- **out_len** length of output buffer

Return: 0 Returned upon successfully computing the HMAC-BLAKE2s MAC.

Example

```
int ret;
byte mac[WC_BLAKE2S_DIGEST_SIZE];
byte data[] = {1, 2, 3};
byte key[] = {4, 5, 6};
ret = wc_Blake2sHmac(data, sizeof(data), key, sizeof(key), mac, sizeof(mac));
if (ret != 0) {
    // error generating HMAC-BLAKE2s
}
```

C.6.3 Source code

```
int wc_InitBlake2b(Blake2b* b2b, word32 digestSz);

int wc_Blake2bUpdate(Blake2b* b2b, const byte* data, word32 sz);

int wc_Blake2bFinal(Blake2b* b2b, byte* final, word32 requestSz);

int wc_Blake2bHmacInit(Blake2b * b2b,
        const byte * key, size_t key_len);

int wc_Blake2bHmacUpdate(Blake2b * b2b,
        const byte * in, size_t in_len);

int wc_Blake2bHmacFinal(Blake2b * b2b,
        const byte * key, size_t key_len,
        byte * out, size_t out_len);

int wc_Blake2bHmac(const byte * in, size_t in_len,
        const byte * key, size_t key_len,
        byte * out, size_t out_len);


int wc_InitBlake2s(Blake2s* b2s, word32 digestSz);

int wc_Blake2sUpdate(Blake2s* b2s, const byte* data, word32 sz);

int wc_Blake2sFinal(Blake2s* b2s, byte* final, word32 requestSz);

int wc_Blake2sHmacInit(Blake2s * b2s,
        const byte * key, size_t key_len);

int wc_Blake2sHmacUpdate(Blake2s * b2s,
        const byte * in, size_t in_len);

int wc_Blake2sHmacFinal(Blake2s * b2s,
        const byte * key, size_t key_len,
        byte * out, size_t out_len);

int wc_Blake2sHmac(const byte * in, size_t in_len,
        const byte * key, size_t key_len,
```

```
        byte * out, size_t out_len);
```

C.7 bn.h

C.7.1 Functions

	Name
int	**wolfSSL_BN_mod_exp**(WOLFSSL_BIGNUM * r, const WOLFSSL_BIGNUM * a, const WOLFSSL_BIGNUM * p, const WOLFSSL_BIGNUM * m, WOLFSSL_BN_CTX * ctx)This function performs the following math “r = (a^p) % m”.

C.7.2 Functions Documentation

C.7.2.1 function wolfSSL_BN_mod_exp

```
int wolfSSL_BN_mod_exp(
    WOLFSSL_BIGNUM * r,
    const WOLFSSL_BIGNUM * a,
    const WOLFSSL_BIGNUM * p,
    const WOLFSSL_BIGNUM * m,
    WOLFSSL_BN_CTX * ctx
)
```

This function performs the following math “r = (a^p) % m”.

Parameters:

- **r** structure to hold result.
- **a** value to be raised by a power.
- **p** power to raise a by.
- **m** modulus to use.
- **ctx** currently not used with wolfSSL can be NULL.

See:

- wolfSSL_BN_new
- wolfSSL_BN_free

Return:

- SSL_SUCCESS On successfully performing math operation.
- SSL_FAILURE If an error case was encountered.

Example

```
WOLFSSL_BIGNUM r,a,p,m;
int ret;
// set big number values
ret  = wolfSSL_BN_mod_exp(r, a, p, m, NULL);
// check ret value
```

C.7.3 Source code

```
int wolfSSL_BN_mod_exp(WOLFSSL_BIGNUM *r, const WOLFSSL_BIGNUM *a,
        const WOLFSSL_BIGNUM *p, const WOLFSSL_BIGNUM *m, WOLFSSL_BN_CTX *ctx);
```

C.8 camellia.h

C.8.1 Functions

	Name
int	**wc_CamelliaSetKey**(wc_Camellia * cam, const byte * key, word32 len, const byte * iv)This function sets the key and initialization vector for a camellia object, initializing it for use as a cipher.
int	**wc_CamelliaSetIV**(wc_Camellia * cam, const byte * iv)This function sets the initialization vector for a camellia object.
int	**wc_CamelliaEncryptDirect**(wc_Camellia * cam, byte * out, const byte * in)This function does a one-block encrypt using the provided camellia object. It parses the first 16 byte block from the buffer in and stores the encrypted result in the buffer out. Before using this function, one should initialize the camellia object using wc_CamelliaSetKey.
int	**wc_CamelliaDecryptDirect**(wc_Camellia * cam, byte * out, const byte * in)This function does a one-block decrypt using the provided camellia object. It parses the first 16 byte block from the buffer in, decrypts it, and stores the result in the buffer out. Before using this function, one should initialize the camellia object using wc_CamelliaSetKey.
int	**wc_CamelliaCbcEncrypt**(wc_Camellia * cam, byte * out, const byte * in, word32 sz)This function encrypts the plaintext from the buffer in and stores the output in the buffer out. It performs this encryption using Camellia with Cipher Block Chaining (CBC).
int	**wc_CamelliaCbcDecrypt**(wc_Camellia * cam, byte * out, const byte * in, word32 sz)This function decrypts the ciphertext from the buffer in and stores the output in the buffer out. It performs this decryption using Camellia with Cipher Block Chaining (CBC).

C.8.2 Functions Documentation

C.8.2.1 function wc_CamelliaSetKey

```
int wc_CamelliaSetKey(
    wc_Camellia * cam,
    const byte * key,
    word32 len,
    const byte * iv
)
```

This function sets the key and initialization vector for a camellia object, initializing it for use as a cipher.

Parameters:

- **cam** pointer to the camellia structure on which to set the key and iv
- **key** pointer to the buffer containing the 16, 24, or 32 byte key to use for encryption and decryption
- **len** length of the key passed in
- **iv** pointer to the buffer containing the 16 byte initialization vector for use with this camellia structure

See:

- wc_CamelliaEncryptDirect
- wc_CamelliaDecryptDirect
- wc_CamelliaCbcEncrypt
- wc_CamelliaCbcDecrypt

Return:

- 0 Returned upon successfully setting the key and initialization vector
- BAD_FUNC_ARG returned if there is an error processing one of the input arguments
- MEMORY_E returned if there is an error allocating memory with XMALLOC

Example

```
Camellia cam;
byte key[32];
// initialize key
byte iv[16];
// initialize iv
if( wc_CamelliaSetKey(&cam, key, sizeof(key), iv) != 0) {
    // error initializing camellia structure
}
```

C.8.2.2 function wc_CamelliaSetIV

```
int wc_CamelliaSetIV(
    wc_Camellia * cam,
    const byte * iv
)
```

This function sets the initialization vector for a camellia object.

Parameters:

- **cam** pointer to the camellia structure on which to set the iv
- **iv** pointer to the buffer containing the 16 byte initialization vector for use with this camellia structure

See: wc_CamelliaSetKey

Return:

- 0 Returned upon successfully setting the key and initialization vector

- BAD_FUNC_ARG returned if there is an error processing one of the input arguments

Example

```
Camellia cam;
byte iv[16];
// initialize iv
if( wc_CamelliaSetIV(&cam, iv) != 0) {
// error initializing camellia structure
}
```

C.8.2.3 function wc_CamelliaEncryptDirect

```
int wc_CamelliaEncryptDirect(
    wc_Camellia * cam,
    byte * out,
    const byte * in
)
```

This function does a one-block encrypt using the provided camellia object. It parses the first 16 byte block from the buffer in and stores the encrypted result in the buffer out. Before using this function, one should initialize the camellia object using wc_CamelliaSetKey.

Parameters:

- **cam** pointer to the camellia structure to use for encryption
- **out** pointer to the buffer in which to store the encrypted block
- **in** pointer to the buffer containing the plaintext block to encrypt

See: wc_CamelliaDecryptDirect

Return: none No returns.

Example

```
Camellia cam;
// initialize cam structure with key and iv
byte plain[] = { // initialize with message to encrypt };
byte cipher[16];

wc_CamelliaEncryptDirect(&ca, cipher, plain);
```

C.8.2.4 function wc_CamelliaDecryptDirect

```
int wc_CamelliaDecryptDirect(
    wc_Camellia * cam,
    byte * out,
    const byte * in
)
```

This function does a one-block decrypt using the provided camellia object. It parses the first 16 byte block from the buffer in, decrypts it, and stores the result in the buffer out. Before using this function, one should initialize the camellia object using wc_CamelliaSetKey.

Parameters:

- **cam** pointer to the camellia structure to use for encryption
- **out** pointer to the buffer in which to store the decrypted plaintext block
- **in** pointer to the buffer containing the ciphertext block to decrypt

See: wc_CamelliaEncryptDirect

Return: none No returns.

Example

```
Camellia cam;
// initialize cam structure with key and iv
byte cipher[] = { // initialize with encrypted message to decrypt };
byte decrypted[16];

wc_CamelliaDecryptDirect(&cam, decrypted, cipher);
```

C.8.2.5 function wc_CamelliaCbcEncrypt

```
int wc_CamelliaCbcEncrypt(
    wc_Camellia * cam,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function encrypts the plaintext from the buffer in and stores the output in the buffer out. It performs this encryption using Camellia with Cipher Block Chaining (CBC).

Parameters:

- **cam** pointer to the camellia structure to use for encryption
- **out** pointer to the buffer in which to store the encrypted ciphertext
- **in** pointer to the buffer containing the plaintext to encrypt
- **sz** the size of the message to encrypt

See: wc_CamelliaCbcDecrypt

Return: none No returns.

Example

```
Camellia cam;
// initialize cam structure with key and iv
byte plain[] = { // initialize with encrypted message to decrypt };
byte cipher[sizeof(plain)];

wc_CamelliaCbcEncrypt(&cam, cipher, plain, sizeof(plain));
```

C.8.2.6 function wc_CamelliaCbcDecrypt

```
int wc_CamelliaCbcDecrypt(
    wc_Camellia * cam,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function decrypts the ciphertext from the buffer in and stores the output in the buffer out. It performs this decryption using Camellia with Cipher Block Chaining (CBC).

Parameters:

- **cam** pointer to the camellia structure to use for encryption
- **out** pointer to the buffer in which to store the decrypted message

- **in** pointer to the buffer containing the encrypted ciphertext
- **sz** the size of the message to encrypt

See: wc_CamelliaCbcEncrypt

Return: none No returns.

Example

```
Camellia cam;
// initialize cam structure with key and iv
byte cipher[] = { // initialize with encrypted message to decrypt };
byte decrypted[sizeof(cipher)];

wc_CamelliaCbcDecrypt(&cam, decrypted, cipher, sizeof(cipher));
```

C.8.3 Source code

```
int  wc_CamelliaSetKey(wc_Camellia* cam, const byte* key, word32 len,
                                    const byte* iv);

int  wc_CamelliaSetIV(wc_Camellia* cam, const byte* iv);

int  wc_CamelliaEncryptDirect(wc_Camellia* cam, byte* out,
                                                          const byte* in);

int  wc_CamelliaDecryptDirect(wc_Camellia* cam, byte* out,
                                                          const byte* in);

int wc_CamelliaCbcEncrypt(wc_Camellia* cam,
                                       byte* out, const byte* in, word32 sz);

int wc_CamelliaCbcDecrypt(wc_Camellia* cam,
                                       byte* out, const byte* in, word32 sz);
```

C.9 chacha20_poly1305.h

C.9.1 Functions

	Name
int	**wc_ChaCha20Poly1305_Encrypt**(const byte inKey[CHACHA20_POLY1305_AEAD_KEYSIZE], const byte inIV[CHACHA20_POLY1305_AEAD_IV_SIZE], const byte * inAAD, word32 inAADLen, const byte * inPlaintext, word32 inPlaintextLen, byte * outCiphertext, byte outAuth-Tag[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE])This function encrypts an input message, inPlaintext, using the ChaCha20 stream cipher, into the output buffer, outCiphertext. It also performs Poly_1305 authentication (on the cipher text), and stores the generated authentication tag in the output buffer, outAuthTag.
int	**wc_ChaCha20Poly1305_Decrypt**(const byte inKey[CHACHA20_POLY1305_AEAD_KEYSIZE], const byte inIV[CHACHA20_POLY1305_AEAD_IV_SIZE], const byte * inAAD, word32 inAADLen, const byte * inCiphertext, word32 inCiphertextLen, const byte inAuth-Tag[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE], byte * outPlaintext)This function decrypts input ciphertext, inCiphertext, using the ChaCha20 stream cipher, into the output buffer, outPlaintext. It also performs Poly_1305 authentication, comparing the given inAuthTag to an authentication generated with the inAAD (arbitrary length additional authentication data). If a nonzero error code is returned, the output data, outPlaintext, is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.

COPYRIGHT ©2024 wolfSSL Inc.

	Name
int	**wc_ChaCha20Poly1305_Encrypt_ex**(ChaCha * chacha, Poly1305 * poly, byte * out, const byte * in, word32 sz, const byte * nonce, byte * tag, const byte * aad, word32 aadSz)This function performs the same AEAD encryption as wc_ChaCha20Poly1305_Encrypt, but takes a caller_owned ChaCha20 context whose key has already been set (with wc_Chacha_SetKey) plus a Poly1305 context, instead of a raw key. This lets the ChaCha20 key be set once and reused across many records, varying only the per-record nonce - it is intended for protocol record layers such as TLS. A fresh per-record Poly1305 key is derived from the ChaCha20 keystream, so this call re-keys the supplied Poly1305 context; the caller does not key it. The 16 byte authentication tag over the AAD and ciphertext is written to tag.
int	**wc_ChaCha20Poly1305_Decrypt_ex**(ChaCha * chacha, Poly1305 * poly, byte * out, const byte * in, word32 sz, const byte * nonce, const byte * tag, const byte * aad, word32 aadSz)This function is the decryption counterpart of wc_ChaCha20Poly1305_Encrypt_ex. It takes a caller_owned ChaCha20 context whose key has already been set (with wc_Chacha_SetKey) plus a Poly1305 context, decrypts in to out, and verifies the Poly1305 tag over the AAD and ciphertext. On tag mismatch it returns MAC_CMP_FAILED_E and zeroizes the output buffer, so no unauthenticated plaintext is released. The ChaCha20 key is reused across records, varying only the per_record nonce; the Poly1305 context is re_keyed internally on each call. out may alias in (in_place decryption is supported).
int	**wc_ChaCha20Poly1305_CheckTag**(const byte authTag[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE], const byte authTagChk[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE])Compares two authentication tags in constant time to prevent timing attacks.
int	**wc_ChaCha20Poly1305_Init**(ChaChaPoly_Aead * aead, const byte inKey[CHACHA20_POLY1305_AEAD_KEYSIZE], const byte inIV[CHACHA20_POLY1305_AEAD_IV_SIZE], int isEncrypt)Initializes a ChaChaPoly_Aead structure for incremental encryption or decryption operations.

	Name
int	**wc_ChaCha20Poly1305_UpdateAad**(ChaChaPoly_Aead * aead, const byte * inAAD, word32 inAADLen)Updates the AEAD context with additional authenticated data (AAD). Must be called after Init and before UpdateData.
int	**wc_ChaCha20Poly1305_UpdateData**(ChaChaPoly_Aead * aead, const byte * inData, byte * outData, word32 dataLen)Encrypts or decrypts data incrementally. Can be called multiple times to process data in chunks.
int	**wc_ChaCha20Poly1305_Final**(ChaChaPoly_Aead * aead, byte outAuth-Tag[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE])Finalizes the AEAD operation and generates the authentication tag.
int	**wc_XChaCha20Poly1305_Init**(ChaChaPoly_Aead * aead, const byte * ad, word32 ad_len, const byte * inKey, word32 inKeySz, const byte * inIV, word32 inIVSz, int isEncrypt)Initializes XChaCha20-Poly1305 AEAD with extended nonce. XChaCha20 uses a 24-byte nonce instead of 12-byte.
int	**wc_XChaCha20Poly1305_Encrypt**(byte * dst, size_t dst_space, const byte * src, size_t src_len, const byte * ad, size_t ad_len, const byte * nonce, size_t nonce_len, const byte * key, size_t key_len)One-shot XChaCha20-Poly1305 encryption with 24-byte nonce.
int	**wc_XChaCha20Poly1305_Decrypt**(byte * dst, size_t dst_space, const byte * src, size_t src_len, const byte * ad, size_t ad_len, const byte * nonce, size_t nonce_len, const byte * key, size_t key_len)One-shot XChaCha20-Poly1305 decryption with 24-byte nonce.

C.9.2 Functions Documentation

C.9.2.1 function wc_ChaCha20Poly1305_Encrypt

```
int wc_ChaCha20Poly1305_Encrypt(
    const byte inKey[CHACHA20_POLY1305_AEAD_KEYSIZE],
    const byte inIV[CHACHA20_POLY1305_AEAD_IV_SIZE],
    const byte * inAAD,
    word32 inAADLen,
    const byte * inPlaintext,
    word32 inPlaintextLen,
    byte * outCiphertext,
    byte outAuthTag[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE]
)
```

This function encrypts an input message, inPlaintext, using the ChaCha20 stream cipher, into the output buffer, outCiphertext. It also performs Poly-1305 authentication (on the cipher text), and stores the generated authentication tag in the output buffer, outAuthTag.

Parameters:

- **inKey** pointer to a buffer containing the 32 byte key to use for encryption
- **inIv** pointer to a buffer containing the 12 byte iv to use for encryption
- **inAAD** pointer to the buffer containing arbitrary length additional authenticated data (AAD)
- **inAADLen** length of the input AAD
- **inPlaintext** pointer to the buffer containing the plaintext to encrypt
- **inPlaintextLen** the length of the plain text to encrypt
- **outCiphertext** pointer to the buffer in which to store the ciphertext
- **outAuthTag** pointer to a 16 byte wide buffer in which to store the authentication tag

See:

- wc_ChaCha20Poly1305_Decrypt
- wc_ChaCha_*
- wc_Poly1305*

Return:

- 0 Returned upon successfully encrypting the message
- BAD_FUNC_ARG returned if there is an error during the encryption process

Example

```
byte key[] = { // initialize 32 byte key };
byte iv[]  = { // initialize 12 byte key };
byte inAAD[] = { // initialize AAD };

byte plain[] = { // initialize message to encrypt };
byte cipher[sizeof(plain)];
byte authTag[16];

int ret = wc_ChaCha20Poly1305_Encrypt(key, iv, inAAD, sizeof(inAAD),
plain, sizeof(plain), cipher, authTag);

if(ret != 0) {
    // error running encrypt
}
```

C.9.2.2 function wc_ChaCha20Poly1305_Decrypt

```
int wc_ChaCha20Poly1305_Decrypt(
    const byte inKey[CHACHA20_POLY1305_AEAD_KEYSIZE],
    const byte inIV[CHACHA20_POLY1305_AEAD_IV_SIZE],
    const byte * inAAD,
    word32 inAADLen,
    const byte * inCiphertext,
    word32 inCiphertextLen,
    const byte inAuthTag[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE],
    byte * outPlaintext
)
```

This function decrypts input ciphertext, inCiphertext, using the ChaCha20 stream cipher, into the output buffer, outPlaintext. It also performs Poly-1305 authentication, comparing the given inAuthTag to an authentication generated with the inAAD (arbitrary length additional authentication data). If a nonzero error code is returned, the output data, outPlaintext, is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.

Parameters:

- **inKey** pointer to a buffer containing the 32 byte key to use for decryption
- **inIv** pointer to a buffer containing the 12 byte iv to use for decryption
- **inAAD** pointer to the buffer containing arbitrary length additional authenticated data (AAD)
- **inAADLen** length of the input AAD
- **inCiphertext** pointer to the buffer containing the ciphertext to decrypt
- **outCiphertextLen** the length of the ciphertext to decrypt
- **inAuthTag** pointer to the buffer containing the 16 byte digest for authentication
- **outPlaintext** pointer to the buffer in which to store the plaintext

See:

- wc_ChaCha20Poly1305_Encrypt
- wc_ChaCha_*
- wc_Poly1305*

Return:

- 0 Returned upon successfully decrypting and authenticating the message
- BAD_FUNC_ARG Returned if any of the function arguments do not match what is expected
- MAC_CMP_FAILED_E Returned if the generated authentication tag does not match the supplied inAuthTag.
- MEMORY_E Returned if internal buffer allocation failed.
- CHACHA_POLY_OVERFLOW Can be returned if input is corrupted.

Example

```
byte key[]   = { // initialize 32 byte key };
byte iv[]    = { // initialize 12 byte key };
byte inAAD[] = { // initialize AAD };

byte cipher[]    = { // initialize with received ciphertext };
byte authTag[16] = { // initialize with received authentication tag };

byte plain[sizeof(cipher)];

int ret = wc_ChaCha20Poly1305_Decrypt(key, iv, inAAD, sizeof(inAAD),
cipher, sizeof(cipher), authTag, plain);

if(ret == MAC_CMP_FAILED_E) {
    // error during authentication
} else if( ret != 0) {
    // error with function arguments
}
```

C.9.2.3 function wc_ChaCha20Poly1305_Encrypt_ex

```
int wc_ChaCha20Poly1305_Encrypt_ex(
    ChaCha * chacha,
    Poly1305 * poly,
    byte * out,
    const byte * in,
    word32 sz,
    const byte * nonce,
    byte * tag,
    const byte * aad,
    word32 aadSz
)
```

This function performs the same AEAD encryption as wc_ChaCha20Poly1305_Encrypt, but takes a caller-owned ChaCha20 context whose key has already been set (with wc_Chacha_SetKey) plus a Poly1305 context, instead of a raw key. This lets the ChaCha20 key be set once and reused across many records, varying only the per-record nonce - it is intended for protocol record layers such as TLS. A fresh per-record Poly1305 key is derived from the ChaCha20 keystream, so this call re-keys the supplied Poly1305 context; the caller does not key it. The 16 byte authentication tag over the AAD and ciphertext is written to tag.

Parameters:

- **chacha** pointer to a ChaCha20 context already keyed with wc_Chacha_SetKey
- **poly** pointer to a Poly1305 context used for the per-record MAC; it is re-keyed internally on each call
- **out** pointer to the buffer in which to store the ciphertext (sz bytes)
- **in** pointer to the buffer containing the plaintext to encrypt
- **sz** the length in bytes of the plaintext to encrypt
- **nonce** pointer to the 12 byte per-record nonce
- **tag** pointer to a 16 byte buffer in which to store the authentication tag
- **aad** pointer to the buffer containing arbitrary length additional authenticated data (AAD)
- **aadSz** length of the input AAD

See:

- wc_ChaCha20Poly1305_Decrypt_ex
- wc_ChaCha20Poly1305_Encrypt
- wc_Chacha_SetKey

Return:

- 0 Returned upon successfully encrypting the message
- BAD_FUNC_ARG Returned if a required pointer argument is NULL (with its matching length nonzero) or otherwise invalid

Example

```
ChaCha chacha;
Poly1305 poly;
byte key[]   = { // initialize 32 byte key };
byte nonce[] = { // initialize 12 byte per-record nonce };
byte aad[]   = { // initialize AAD };
byte plain[] = { // initialize message to encrypt };
byte cipher[sizeof(plain)];
byte authTag[16];

wc_Chacha_SetKey(&chacha, key, sizeof(key));   // once, then reuse
int ret = wc_ChaCha20Poly1305_Encrypt_ex(&chacha, &poly, cipher, plain,
    sizeof(plain), nonce, authTag, aad, sizeof(aad));
if (ret != 0) {
    // error running encrypt
}
```

C.9.2.4 function wc_ChaCha20Poly1305_Decrypt_ex

```
int wc_ChaCha20Poly1305_Decrypt_ex(
    ChaCha * chacha,
    Poly1305 * poly,
    byte * out,
    const byte * in,
```

```
    word32 sz,
    const byte * nonce,
    const byte * tag,
    const byte * aad,
    word32 aadSz
)
```

This function is the decryption counterpart of wc_ChaCha20Poly1305_Encrypt_ex. It takes a caller-owned ChaCha20 context whose key has already been set (with wc_Chacha_SetKey) plus a Poly1305 context, decrypts in to out, and verifies the Poly1305 tag over the AAD and ciphertext. On tag mismatch it returns MAC_CMP_FAILED_E and zeroizes the output buffer, so no unauthenticated plaintext is released. The ChaCha20 key is reused across records, varying only the per-record nonce; the Poly1305 context is re-keyed internally on each call. out may alias in (in-place decryption is supported).

Parameters:

- **chacha** pointer to a ChaCha20 context already keyed with wc_Chacha_SetKey
- **poly** pointer to a Poly1305 context used for the per-record MAC; it is re-keyed internally on each call
- **out** pointer to the buffer in which to store the plaintext (sz bytes)
- **in** pointer to the buffer containing the ciphertext to decrypt
- **sz** the length in bytes of the ciphertext to decrypt
- **nonce** pointer to the 12 byte per-record nonce
- **tag** pointer to the 16 byte authentication tag to verify
- **aad** pointer to the buffer containing arbitrary length additional authenticated data (AAD)
- **aadSz** length of the input AAD

See:

- wc_ChaCha20Poly1305_Encrypt_ex
- wc_ChaCha20Poly1305_Decrypt
- wc_Chacha_SetKey

Return:

- 0 Returned upon successfully decrypting and authenticating the message
- MAC_CMP_FAILED_E Returned if the computed authentication tag does not match the supplied tag; out is zeroized in this case
- BAD_FUNC_ARG Returned if a required pointer argument is NULL (with its matching length nonzero) or otherwise invalid

Example

```
ChaCha chacha;
Poly1305 poly;
byte key[]       = { // initialize 32 byte key };
byte nonce[]     = { // initialize 12 byte per-record nonce };
byte aad[]       = { // initialize AAD };
byte cipher[]    = { // received ciphertext };
byte authTag[16] = { // received authentication tag };
byte plain[sizeof(cipher)];

wc_Chacha_SetKey(&chacha, key, sizeof(key));   // once, then reuse
int ret = wc_ChaCha20Poly1305_Decrypt_ex(&chacha, &poly, plain, cipher,
    sizeof(cipher), nonce, authTag, aad, sizeof(aad));
if (ret == MAC_CMP_FAILED_E) {
    // authentication failed; plain has been zeroized
} else if (ret != 0) {
```

```
    // error with function arguments
}
```

C.9.2.5 function wc_ChaCha20Poly1305_CheckTag

```
int wc_ChaCha20Poly1305_CheckTag(
    const byte authTag[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE],
    const byte authTagChk[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE]
)
```

Compares two authentication tags in constant time to prevent timing attacks.

Parameters:

- **authTag** First authentication tag
- **authTagChk** Second authentication tag to compare

See: wc_ChaCha20Poly1305_Decrypt

Return:

- 0 If tags match
- MAC_CMP_FAILED_E If tags do not match

Example

```
byte tag1[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE];
byte tag2[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE];

int ret = wc_ChaCha20Poly1305_CheckTag(tag1, tag2);
if (ret != 0) {
    // tags do not match
}
```

C.9.2.6 function wc_ChaCha20Poly1305_Init

```
int wc_ChaCha20Poly1305_Init(
    ChaChaPoly_Aead * aead,
    const byte inKey[CHACHA20_POLY1305_AEAD_KEYSIZE],
    const byte inIV[CHACHA20_POLY1305_AEAD_IV_SIZE],
    int isEncrypt
)
```

Initializes a ChaChaPoly_Aead structure for incremental encryption or decryption operations.

Parameters:

- **aead** Pointer to ChaChaPoly_Aead structure to initialize
- **inKey** 32-byte encryption key
- **inIV** 12-byte initialization vector
- **isEncrypt** 1 for encryption, 0 for decryption

See:

- wc_ChaCha20Poly1305_UpdateAad
- wc_ChaCha20Poly1305_UpdateData
- wc_ChaCha20Poly1305_Final

Return:

- 0 On success
- BAD_FUNC_ARG If parameters are invalid

Example

```
ChaChaPoly_Aead aead;
byte key[CHACHA20_POLY1305_AEAD_KEYSIZE];
byte iv[CHACHA20_POLY1305_AEAD_IV_SIZE];

int ret = wc_ChaCha20Poly1305_Init(&aead, key, iv, 1);
if (ret != 0) {
    // error initializing
}
```

C.9.2.7 function wc_ChaCha20Poly1305_UpdateAad

```
int wc_ChaCha20Poly1305_UpdateAad(
    ChaChaPoly_Aead * aead,
    const byte * inAAD,
    word32 inAADLen
)
```

Updates the AEAD context with additional authenticated data (AAD). Must be called after Init and before UpdateData.

Parameters:

- **aead** Pointer to initialized ChaChaPoly_Aead structure
- **inAAD** Additional authenticated data
- **inAADLen** Length of AAD in bytes

See:

- wc_ChaCha20Poly1305_Init
- wc_ChaCha20Poly1305_UpdateData

Return:

- 0 On success
- BAD_FUNC_ARG If parameters are invalid

Example

```
ChaChaPoly_Aead aead;
byte aad[]; // AAD data

wc_ChaCha20Poly1305_Init(&aead, key, iv, 1);
int ret = wc_ChaCha20Poly1305_UpdateAad(&aead, aad, sizeof(aad));
if (ret != 0) {
    // error updating AAD
}
```

C.9.2.8 function wc_ChaCha20Poly1305_UpdateData

```
int wc_ChaCha20Poly1305_UpdateData(
    ChaChaPoly_Aead * aead,
    const byte * inData,
    byte * outData,
    word32 dataLen
)
```

Encrypts or decrypts data incrementally. Can be called multiple times to process data in chunks.

Parameters:

- **aead** Pointer to initialized ChaChaPoly_Aead structure
- **inData** Input data (plaintext or ciphertext)
- **outData** Output buffer for result
- **dataLen** Length of data to process

See:

- wc_ChaCha20Poly1305_Init
- wc_ChaCha20Poly1305_Final

Return:

- 0 On success
- BAD_FUNC_ARG If parameters are invalid

Example

```
ChaChaPoly_Aead aead;
byte plain[]; // plaintext
byte cipher[sizeof(plain)];

wc_ChaCha20Poly1305_Init(&aead, key, iv, 1);
wc_ChaCha20Poly1305_UpdateAad(&aead, aad, aadLen);
int ret = wc_ChaCha20Poly1305_UpdateData(&aead, plain,
                                         cipher, sizeof(plain));
```

C.9.2.9 function wc_ChaCha20Poly1305_Final

```
int wc_ChaCha20Poly1305_Final(
    ChaChaPoly_Aead * aead,
    byte outAuthTag[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE]
)
```

Finalizes the AEAD operation and generates the authentication tag.

Parameters:

- **aead** Pointer to ChaChaPoly_Aead structure
- **outAuthTag** Buffer to store 16-byte authentication tag

See:

- wc_ChaCha20Poly1305_Init
- wc_ChaCha20Poly1305_UpdateData

Return:

- 0 On success
- BAD_FUNC_ARG If parameters are invalid

Example

```
ChaChaPoly_Aead aead;
byte authTag[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE];

wc_ChaCha20Poly1305_Init(&aead, key, iv, 1);
wc_ChaCha20Poly1305_UpdateAad(&aead, aad, aadLen);
wc_ChaCha20Poly1305_UpdateData(&aead, plain, cipher, plainLen);
int ret = wc_ChaCha20Poly1305_Final(&aead, authTag);
```

C.9.2.10 function wc_XChaCha20Poly1305_Init

```
int wc_XChaCha20Poly1305_Init(
    ChaChaPoly_Aead * aead,
    const byte * ad,
    word32 ad_len,
    const byte * inKey,
    word32 inKeySz,
    const byte * inIV,
    word32 inIVSz,
    int isEncrypt
)
```

Initializes XChaCha20-Poly1305 AEAD with extended nonce. XChaCha20 uses a 24-byte nonce instead of 12-byte.

Parameters:

- **aead** Pointer to ChaChaPoly_Aead structure
- **ad** Additional authenticated data
- **ad_len** Length of AAD
- **inKey** Encryption key
- **inKeySz** Key size (must be 32)
- **inIV** Initialization vector
- **inIVSz** IV size (must be 24 for XChaCha20)
- **isEncrypt** 1 for encryption, 0 for decryption

See:

- wc_XChaCha20Poly1305_Encrypt
- wc_XChaCha20Poly1305_Decrypt

Return:

- 0 On success
- BAD_FUNC_ARG If parameters are invalid

Example

```
ChaChaPoly_Aead aead;
byte key[32];
byte iv[24];
byte aad[]; // AAD

int ret = wc_XChaCha20Poly1305_Init(&aead, aad, sizeof(aad),
                                    key, 32, iv, 24, 1);
```

C.9.2.11 function wc_XChaCha20Poly1305_Encrypt

```
int wc_XChaCha20Poly1305_Encrypt(
    byte * dst,
    size_t dst_space,
    const byte * src,
    size_t src_len,
    const byte * ad,
    size_t ad_len,
    const byte * nonce,
    size_t nonce_len,
    const byte * key,
```

```
    size_t key_len
)
```

One-shot XChaCha20-Poly1305 encryption with 24-byte nonce.

Parameters:

- **dst** Output buffer for ciphertext and tag
- **dst_space** Size of output buffer
- **src** Input plaintext
- **src_len** Length of plaintext
- **ad** Additional authenticated data
- **ad_len** Length of AAD
- **nonce** 24-byte nonce
- **nonce_len** Nonce length (must be 24)
- **key** 32-byte encryption key
- **key_len** Key length (must be 32)

See: wc_XChaCha20Poly1305_Decrypt

Return:

- 0 On success
- BAD_FUNC_ARG If parameters are invalid
- BUFFER_E If dst_space is insufficient

Example

```
byte key[32], nonce[24];
byte plain[]; // plaintext
byte cipher[sizeof(plain) + 16];

int ret = wc_XChaCha20Poly1305_Encrypt(cipher, sizeof(cipher),
                                       plain, sizeof(plain),
                                       NULL, 0, nonce, 24,
                                       key, 32);
```

C.9.2.12 function wc_XChaCha20Poly1305_Decrypt

```
int wc_XChaCha20Poly1305_Decrypt(
    byte * dst,
    size_t dst_space,
    const byte * src,
    size_t src_len,
    const byte * ad,
    size_t ad_len,
    const byte * nonce,
    size_t nonce_len,
    const byte * key,
    size_t key_len
)
```

One-shot XChaCha20-Poly1305 decryption with 24-byte nonce.

Parameters:

- **dst** Output buffer for plaintext
- **dst_space** Size of output buffer
- **src** Input ciphertext with tag
- **src_len** Length of ciphertext plus tag

- **ad** Additional authenticated data
- **ad_len** Length of AAD
- **nonce** 24-byte nonce
- **nonce_len** Nonce length (must be 24)
- **key** 32-byte decryption key
- **key_len** Key length (must be 32)

See: wc_XChaCha20Poly1305_Encrypt

Return:

- 0 On success
- BAD_FUNC_ARG If parameters are invalid
- BUFFER_E If dst_space is insufficient
- MAC_CMP_FAILED_E If authentication fails

Example

```
byte key[32], nonce[24];
byte cipher[]; // ciphertext + tag
byte plain[sizeof(cipher) - 16];

int ret = wc_XChaCha20Poly1305_Decrypt(plain, sizeof(plain),
                                       cipher, sizeof(cipher),
                                       NULL, 0, nonce, 24,
                                       key, 32);
if (ret == MAC_CMP_FAILED_E) {
    // authentication failed
}
```

C.9.3 Source code

```
int wc_ChaCha20Poly1305_Encrypt(
                const byte inKey[CHACHA20_POLY1305_AEAD_KEYSIZE],
                const byte inIV[CHACHA20_POLY1305_AEAD_IV_SIZE],
                const byte* inAAD, word32 inAADLen,
                const byte* inPlaintext, word32 inPlaintextLen,
                byte* outCiphertext,
                byte outAuthTag[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE]);

int wc_ChaCha20Poly1305_Decrypt(
                const byte inKey[CHACHA20_POLY1305_AEAD_KEYSIZE],
                const byte inIV[CHACHA20_POLY1305_AEAD_IV_SIZE],
                const byte* inAAD, word32 inAADLen,
                const byte* inCiphertext, word32 inCiphertextLen,
                const byte inAuthTag[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE],
                byte* outPlaintext);

int wc_ChaCha20Poly1305_Encrypt_ex(ChaCha* chacha, Poly1305* poly,
    byte* out, const byte* in, word32 sz, const byte* nonce, byte* tag,
    const byte* aad, word32 aadSz);

int wc_ChaCha20Poly1305_Decrypt_ex(
    ChaCha* chacha, Poly1305* poly, byte* out, const byte* in, word32 sz,
    const byte* nonce, const byte* tag, const byte* aad, word32 aadSz);
```

```
int wc_ChaCha20Poly1305_CheckTag(
                const byte authTag[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE],
                const byte authTagChk[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE]);

int wc_ChaCha20Poly1305_Init(ChaChaPoly_Aead* aead,
                const byte inKey[CHACHA20_POLY1305_AEAD_KEYSIZE],
                const byte inIV[CHACHA20_POLY1305_AEAD_IV_SIZE],
                int isEncrypt);

int wc_ChaCha20Poly1305_UpdateAad(ChaChaPoly_Aead* aead,
                const byte* inAAD, word32 inAADLen);

int wc_ChaCha20Poly1305_UpdateData(ChaChaPoly_Aead* aead,
                const byte* inData, byte* outData, word32 dataLen);

int wc_ChaCha20Poly1305_Final(ChaChaPoly_Aead* aead,
                byte outAuthTag[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE]);

int wc_XChaCha20Poly1305_Init(ChaChaPoly_Aead* aead,
                const byte *ad, word32 ad_len,
                const byte *inKey, word32 inKeySz,
                const byte *inIV, word32 inIVSz,
                int isEncrypt);

int wc_XChaCha20Poly1305_Encrypt(byte *dst, size_t dst_space,
                const byte *src, size_t src_len,
                const byte *ad, size_t ad_len,
                const byte *nonce, size_t nonce_len,
                const byte *key, size_t key_len);

int wc_XChaCha20Poly1305_Decrypt(byte *dst, size_t dst_space,
                const byte *src, size_t src_len,
                const byte *ad, size_t ad_len,
                const byte *nonce, size_t nonce_len,
                const byte *key, size_t key_len);
```

C.10 chacha.h

C.10.1 Functions

	Name
int	**wc_Chacha_SetIV**(ChaCha * ctx, const byte * inIv, word32 counter)This function sets the initialization vector (nonce) for a ChaCha object, initializing it for use as a cipher. It should be called after the key has been set, using wc_Chacha_SetKey. A difference nonce should be used for each round of encryption.

	Name
int	**wc_Chacha_Process**(ChaCha * ctx, byte * cipher, const byte * plain, word32 msglen)This function processes the text from the buffer input, encrypts or decrypts it, and stores the result in the buffer output.
int	**wc_Chacha_SetKey**(ChaCha * ctx, const byte * key, word32 keySz)This function sets the key for a ChaCha object, initializing it for use as a cipher. It should be called before setting the nonce with wc_Chacha_SetIV, and before using it for encryption with wc_Chacha_Process.
int	**wc_XChacha_SetKey**(ChaCha * ctx, const byte * key, word32 keySz, const byte * nonce, word32 nonceSz, word32 counter)This function sets the key and nonce for an XChaCha cipher context. XChaCha extends ChaCha20 to use a 192-bit nonce instead of 96 bits, providing better security for applications that need to encrypt many messages with the same key.

C.10.2 Functions Documentation

C.10.2.1 function wc_Chacha_SetIV

```
int wc_Chacha_SetIV(
    ChaCha * ctx,
    const byte * inIv,
    word32 counter
)
```

This function sets the initialization vector (nonce) for a ChaCha object, initializing it for use as a cipher. It should be called after the key has been set, using wc_Chacha_SetKey. A difference nonce should be used for each round of encryption.

Parameters:

- **ctx** pointer to the ChaCha structure on which to set the iv
- **inIv** pointer to a buffer containing the 12 byte initialization vector with which to initialize the ChaCha structure
- **counter** the value at which the block counter should start–usually zero.

See:

- wc_Chacha_SetKey
- wc_Chacha_Process

Return:

- 0 Returned upon successfully setting the initialization vector
- BAD_FUNC_ARG returned if there is an error processing the ctx input argument

Example

```
ChaCha enc;
// initialize enc with wc_Chacha_SetKey
byte iv[12];
// initialize iv
```

```
if( wc_Chacha_SetIV(&enc, iv, 0) != 0) {
    // error initializing ChaCha structure
}
```

C.10.2.2 function wc_Chacha_Process

```
int wc_Chacha_Process(
    ChaCha * ctx,
    byte * cipher,
    const byte * plain,
    word32 msglen
)
```

This function processes the text from the buffer input, encrypts or decrypts it, and stores the result in the buffer output.

Parameters:

- **ctx** pointer to the ChaCha structure on which to set the iv
- **output** pointer to a buffer in which to store the output ciphertext or decrypted plaintext
- **input** pointer to the buffer containing the input plaintext to encrypt or the input ciphertext to decrypt
- **msglen** length of the message to encrypt or the ciphertext to decrypt

See:

- wc_Chacha_SetKey
- wc_Chacha_Process

Return:

- 0 Returned upon successfully encrypting or decrypting the input
- BAD_FUNC_ARG returned if there is an error processing the ctx input argument
- MISSING_KEY returned if wc_Chacha_SetKey was not called to set a key before processing

Example

```
ChaCha enc;
// initialize enc with wc_Chacha_SetKey and wc_Chacha_SetIV

byte plain[] = { // initialize plaintext };
byte cipher[sizeof(plain)];
if( wc_Chacha_Process(&enc, cipher, plain, sizeof(plain)) != 0) {
    // error processing ChaCha cipher
}
```

C.10.2.3 function wc_Chacha_SetKey

```
int wc_Chacha_SetKey(
    ChaCha * ctx,
    const byte * key,
    word32 keySz
)
```

This function sets the key for a ChaCha object, initializing it for use as a cipher. It should be called before setting the nonce with wc_Chacha_SetIV, and before using it for encryption with wc_Chacha_Process.

Parameters:

- **ctx** pointer to the ChaCha structure in which to set the key

- **key** pointer to a buffer containing the 16 or 32 byte key with which to initialize the ChaCha structure
- **keySz** the length of the key passed in

See:

- wc_Chacha_SetIV
- wc_Chacha_Process

Return:

- 0 Returned upon successfully setting the key
- BAD_FUNC_ARG returned if there is an error processing the ctx input argument or if the key is not 16 or 32 bytes long

Example

```
ChaCha enc;
byte key[] = { // initialize key };

if( wc_Chacha_SetKey(&enc, key, sizeof(key)) != 0) {
    // error initializing ChaCha structure
}
```

C.10.2.4 function wc_XChacha_SetKey

```
int wc_XChacha_SetKey(
    ChaCha * ctx,
    const byte * key,
    word32 keySz,
    const byte * nonce,
    word32 nonceSz,
    word32 counter
)
```

This function sets the key and nonce for an XChaCha cipher context. XChaCha extends ChaCha20 to use a 192-bit nonce instead of 96 bits, providing better security for applications that need to encrypt many messages with the same key.

Parameters:

- **ctx** pointer to the ChaCha structure to initialize
- **key** pointer to the key buffer (16 or 32 bytes)
- **keySz** length of the key in bytes (16 or 32)
- **nonce** pointer to the nonce buffer (must be 24 bytes)
- **nonceSz** length of the nonce in bytes (must be 24)
- **counter** initial block counter value (usually 0)

See:

- wc_Chacha_SetKey
- wc_Chacha_SetIV
- wc_Chacha_Process

Return:

- 0 On success.
- BAD_FUNC_ARG If ctx, key, or nonce is NULL, or if keySz is invalid, or if nonceSz is not XCHACHA_NONCE_BYTES (24 bytes).
- Other negative values on error.

Example

```
ChaCha ctx;
byte key[32] = { }; // 256-bit key
byte nonce[24] = { }; // 192-bit nonce
byte plaintext[100] = { }; // data to encrypt
byte ciphertext[100];

int ret = wc_XChacha_SetKey(&ctx, key, 32, nonce, 24, 0);
if (ret != 0) {
    // error setting XChaCha key
}
wc_Chacha_Process(&ctx, ciphertext, plaintext, 100);
```

C.10.3 Source code

```
int wc_Chacha_SetIV(ChaCha* ctx, const byte* inIv, word32 counter);

int wc_Chacha_Process(ChaCha* ctx, byte* cipher, const byte* plain,
                              word32 msglen);

int wc_Chacha_SetKey(ChaCha* ctx, const byte* key, word32 keySz);

int wc_XChacha_SetKey(ChaCha *ctx, const byte *key, word32 keySz,
                      const byte *nonce, word32 nonceSz, word32 counter);
```

C.11 cmac.h

C.11.1 Functions

	Name
int	**wc_InitCmac**(Cmac * cmac, const byte * key, word32 keySz, int type, void * unused)Initialize the Cmac structure with defaults.
int	**wc_InitCmac_ex**(Cmac * cmac, const byte * key, word32 keySz, int type, void * unused, void * heap, int devId)Initialize the Cmac structure with defaults.
int	**wc_CmacUpdate**(Cmac * cmac, const byte * in, word32 inSz)Add Cipher-based Message Authentication Code input data.
int	**wc_CmacFinalNoFree**(Cmac * cmac, byte * out, word32 * outSz)Generate the final result using Cipher-based Message Authentication Code, deferring context cleanup.
int	**wc_CmacFinal**(Cmac * cmac, byte * out, word32 * outSz)Generate the final result using Cipher_based Message Authentication Code, and clean up the context with wc_CmacFree().
int	**wc_CmacFree**(Cmac * cmac)Clean up allocations in a CMAC context.

	Name
int	**wc_AesCmacGenerate**(byte * out, word32 * outSz, const byte * in, word32 inSz, const byte * key, word32 keySz)Single shot function for generating a CMAC.
int	**wc_AesCmacVerify**(const byte * check, word32 checkSz, const byte * in, word32 inSz, const byte * key, word32 keySz)Single shot function for validating a CMAC.
int	**wc_CMAC_Grow**(Cmac * cmac, const byte * in, int inSz)Only used with WOLFSSL_HASH_KEEP when hardware requires single-shot and the updates must be cached in memory.
int	**wc_AesCmacGenerate_ex**(Cmac * cmac, byte * out, word32 * outSz, const byte * in, word32 inSz, const byte * key, word32 keySz, void * heap, int devId)Single shot AES-CMAC generation with extended parameters including heap and device ID.
int	**wc_AesCmacVerify_ex**(Cmac * cmac, const byte * check, word32 checkSz, const byte * in, word32 inSz, const byte * key, word32 keySz, void * heap, int devId)Single shot AES-CMAC verification with extended parameters including heap and device ID.
int	**wc_CmacSetTagLen**(Cmac * cmac, word32 tagLen)This function associates one tag length with the key held by a Cmac object, as SP 800-38B section 5.4 requires. Once set, the final and verify calls for that key accept only that tag length, and this call cannot change or clear it. Starting a new Cmac clears the association. The length is kept on the AES object the Cmac already holds.

C.11.2 Functions Documentation

C.11.2.1 function wc_InitCmac

```
int wc_InitCmac(
    Cmac * cmac,
    const byte * key,
    word32 keySz,
    int type,
    void * unused
)
```

Initialize the Cmac structure with defaults.

Parameters:

- **cmac** pointer to the Cmac structure
- **key** key pointer
- **keySz** size of the key pointer (16, 24 or 32)
- **type** Always WC_CMAC_AES = 1

- **unused** not used, exists for potential future use around compatibility

See:

- wc_InitCmac_ex
- wc_CmacUpdate
- wc_CmacFinal
- wc_CmacFinalNoFree
- wc_CmacFree

Return: 0 on success

Example

```
Cmac cmac[1];
ret = wc_InitCmac(cmac, key, keySz, WC_CMAC_AES, NULL);
if (ret == 0) {
    ret = wc_CmacUpdate(cmac, in, inSz);
}
if (ret == 0) {
    ret = wc_CmacFinal(cmac, out, outSz);
}
```

C.11.2.2 function wc_InitCmac_ex

```
int wc_InitCmac_ex(
    Cmac * cmac,
    const byte * key,
    word32 keySz,
    int type,
    void * unused,
    void * heap,
    int devId
)
```

Initialize the Cmac structure with defaults.

Parameters:

- **cmac** pointer to the Cmac structure
- **key** key pointer
- **keySz** size of the key pointer (16, 24 or 32)
- **type** Always WC_CMAC_AES = 1
- **unused** not used, exists for potential future use around compatibility
- **heap** pointer to the heap hint used for dynamic allocation. Typically used with our static memory option. Can be NULL.
- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID (-2) if not used

See:

- wc_InitCmac_ex
- wc_CmacUpdate
- wc_CmacFinal
- wc_CmacFinalNoFree
- wc_CmacFree

Return: 0 on success

Example

```
Cmac cmac[1];
ret = wc_InitCmac_ex(cmac, key, keySz, WC_CMAC_AES, NULL, NULL, INVALID_DEVID);
if (ret == 0) {
    ret = wc_CmacUpdate(cmac, in, inSz);
}
if (ret == 0) {
    ret = wc_CmacFinal(cmac, out, &outSz);
}
```

C.11.2.3 function wc_CmacUpdate

```
int wc_CmacUpdate(
    Cmac * cmac,
    const byte * in,
    word32 inSz
)
```

Add Cipher-based Message Authentication Code input data.

Parameters:

- **cmac** pointer to the Cmac structure
- **in** input data to process
- **inSz** size of input data

See:

- wc_InitCmac
- wc_CmacFinal
- wc_CmacFinalNoFree
- wc_CmacFree

Return: 0 on success

Example

```
ret = wc_CmacUpdate(cmac, in, inSz);
```

C.11.2.4 function wc_CmacFinalNoFree

```
int wc_CmacFinalNoFree(
    Cmac * cmac,
    byte * out,
    word32 * outSz
)
```

Generate the final result using Cipher-based Message Authentication Code, deferring context cleanup.

Parameters:

- **cmac** pointer to the Cmac structure
- **out** pointer to return the result
- **outSz** pointer size of output (in/out)

See:

- wc_InitCmac
- wc_CmacFinal
- wc_CmacFinalNoFree
- wc_CmacFree

Return: 0 on success

Example

```
ret = wc_CmacFinalNoFree(cmac, out, &outSz);
(void)wc_CmacFree(cmac);
```

C.11.2.5 function wc_CmacFinal

```
int wc_CmacFinal(
    Cmac * cmac,
    byte * out,
    word32 * outSz
)
```

Generate the final result using Cipher-based Message Authentication Code, and clean up the context with wc_CmacFree().

Parameters:

- **cmac** pointer to the Cmac structure
- **out** pointer to return the result
- **outSz** pointer size of output (in/out)

See:

- wc_InitCmac
- wc_CmacFinalNoFree
- wc_CmacFinalNoFree
- wc_CmacFree

Return: 0 on success

Example

```
ret = wc_CmacFinal(cmac, out, &outSz);
```

C.11.2.6 function wc_CmacFree

```
int wc_CmacFree(
    Cmac * cmac
)
```

Clean up allocations in a CMAC context.

Parameters:

- **cmac** pointer to the Cmac structure

See:

- wc_InitCmac
- wc_CmacFinalNoFree
- wc_CmacFinal
- wc_CmacFree

Return: 0 on success

Example

```
ret = wc_CmacFinalNoFree(cmac, out, &outSz);
(void)wc_CmacFree(cmac);
```

C.11.2.7 function wc_AesCmacGenerate

```
int wc_AesCmacGenerate(
    byte * out,
    word32 * outSz,
    const byte * in,
    word32 inSz,
    const byte * key,
    word32 keySz
)
```

Single shot function for generating a CMAC.

Parameters:

- **out** pointer to return the result
- **outSz** pointer size of output (in/out)
- **in** input data to process
- **inSz** size of input data
- **key** key pointer
- **keySz** size of the key pointer (16, 24 or 32)

See: wc_AesCmacVerify

Return: 0 on success

Example

```
ret = wc_AesCmacGenerate(mac, &macSz, msg, msgSz, key, keySz);
```

C.11.2.8 function wc_AesCmacVerify

```
int wc_AesCmacVerify(
    const byte * check,
    word32 checkSz,
    const byte * in,
    word32 inSz,
    const byte * key,
    word32 keySz
)
```

Single shot function for validating a CMAC.

Parameters:

- **check** Expected MAC value to verify
- **checkSz** size of expected MAC value; must be in [WC_CMAC_TAG_MIN_SZ, WC_AES_BLOCK_SIZE]
- **in** input data to process
- **inSz** size of input data
- **key** key pointer
- **keySz** size of the key pointer (16, 24 or 32)

See: wc_AesCmacGenerate

Return:

- 0 on success
- BAD_FUNC_ARG if parameters are invalid
- MAC_CMP_FAILED_E if the supplied tag does not match

Example

```
ret = wc_AesCmacVerify(mac, macSz, msg, msgSz, key, keySz);
```

C.11.2.9 function wc_CMAC_Grow

```
int wc_CMAC_Grow(
    Cmac * cmac,
    const byte * in,
    int inSz
)
```

Only used with WOLFSSL_HASH_KEEP when hardware requires single-shot and the updates must be cached in memory.

Parameters:

- **in** input data to process
- **inSz** size of input data

Return: 0 on success

Example

```
ret = wc_CMAC_Grow(cmac, in, inSz)
```

C.11.2.10 function wc_AesCmacGenerate_ex

```
int wc_AesCmacGenerate_ex(
    Cmac * cmac,
    byte * out,
    word32 * outSz,
    const byte * in,
    word32 inSz,
    const byte * key,
    word32 keySz,
    void * heap,
    int devId
)
```

Single shot AES-CMAC generation with extended parameters including heap and device ID.

Parameters:

- **cmac** Pointer to Cmac structure (can be NULL for one-shot)
- **out** Buffer to store MAC output
- **outSz** Pointer to output size (in/out)
- **in** Input data to authenticate
- **inSz** Length of input data
- **key** AES key
- **keySz** Key size (16, 24, or 32 bytes)
- **heap** Heap hint for memory allocation (can be NULL)
- **devId** Device ID for hardware acceleration (use INVALID_DEVID for software)

See:

- wc_AesCmacGenerate
- wc_AesCmacVerify_ex

Return:

- 0 on success
- BAD_FUNC_ARG if parameters are invalid

Example

```
byte mac[AES_BLOCK_SIZE];
word32 macSz = sizeof(mac);
byte key[16], msg[64];

int ret = wc_AesCmacGenerate_ex(NULL, mac, &macSz, msg,
                                sizeof(msg), key, sizeof(key),
                                NULL, INVALID_DEVID);
```

C.11.2.11 function wc_AesCmacVerify_ex

```
int wc_AesCmacVerify_ex(
    Cmac * cmac,
    const byte * check,
    word32 checkSz,
    const byte * in,
    word32 inSz,
    const byte * key,
    word32 keySz,
    void * heap,
    int devId
)
```

Single shot AES-CMAC verification with extended parameters including heap and device ID.

Parameters:

- **cmac** Pointer to Cmac structure
- **check** Expected MAC value to verify
- **checkSz** Size of expected MAC; must be in [WC_CMAC_TAG_MIN_SZ, WC_AES_BLOCK_SIZE]
- **in** Input data to authenticate
- **inSz** Length of input data
- **key** AES key
- **keySz** Key size (16, 24, or 32 bytes)
- **heap** Heap hint for memory allocation (can be NULL)
- **devId** Device ID for hardware acceleration (use INVALID_DEVID for software)

See:

- wc_InitCmac_ex
- wc_AesCmacVerify
- wc_AesCmacGenerate_ex

Return:

- 0 on success
- BAD_FUNC_ARG if parameters are invalid
- MAC_CMP_FAILED_E if MAC verification fails

Example

```
Cmac cmac;
byte mac[WC_AES_BLOCK_SIZE];
byte key[16], msg[64];
int ret;

ret = wc_InitCmac_ex(&cmac, key, sizeof(key), WC_CMAC_AES, NULL,
                     NULL, INVALID_DEVID);
```

```
if (ret == 0) {
    ret = wc_AesCmacVerify_ex(&cmac, mac, sizeof(mac), msg,
                              sizeof(msg), key, sizeof(key),
                              NULL, INVALID_DEVID);
}
if (ret == MAC_CMP_FAILED_E) {
    // MAC verification failed
}
```

C.11.2.12 function wc_CmacSetTagLen

```
int wc_CmacSetTagLen(
    Cmac * cmac,
    word32 tagLen
)
```

This function associates one tag length with the key held by a Cmac object, as SP 800-38B section 5.4 requires. Once set, the final and verify calls for that key accept only that tag length, and this call cannot change or clear it. Starting a new Cmac clears the association. The length is kept on the AES object the Cmac already holds.

Parameters:

- **cmac** pointer to the Cmac object holding the key
- **tagLen** tag length to associate with the key

See:

- wc_InitCmac
- wc_CmacFinal
- wc_AesSetTagLen

Return:

- 0 On success.
- BAD_FUNC_ARG Returned if cmac is NULL, the length is outside WC_CMAC_TAG_MIN_SZ to WC_CMAC_TAG_MAX_SZ, or a different length is already associated.

Example

```
Cmac cmac;
byte key[] = { some 16, 24, 32 byte key };
if (wc_InitCmac(&cmac, key, sizeof(key), WC_CMAC_AES, NULL) != 0) {
    // failed to set up cmac
}
if (wc_CmacSetTagLen(&cmac, 16) != 0) {
    // failed to associate the tag length
}
```

C.11.3 Source code

```
int wc_InitCmac(Cmac* cmac,
                const byte* key, word32 keySz,
                int type, void* unused);

int wc_InitCmac_ex(Cmac* cmac,
                const byte* key, word32 keySz,
                int type, void* unused, void* heap, int devId);
```

```
int wc_CmacUpdate(Cmac* cmac,
                  const byte* in, word32 inSz);


int wc_CmacFinalNoFree(Cmac* cmac,
                 byte* out, word32* outSz);

int wc_CmacFinal(Cmac* cmac,
                 byte* out, word32* outSz);

int wc_CmacFree(Cmac* cmac);

int wc_AesCmacGenerate(byte* out, word32* outSz,
                       const byte* in, word32 inSz,
                       const byte* key, word32 keySz);

int wc_AesCmacVerify(const byte* check, word32 checkSz,
                     const byte* in, word32 inSz,
                     const byte* key, word32 keySz);


int wc_CMAC_Grow(Cmac* cmac, const byte* in, int inSz);

int wc_AesCmacGenerate_ex(Cmac *cmac, byte* out, word32* outSz,
                          const byte* in, word32 inSz,
                          const byte* key, word32 keySz,
                          void* heap, int devId);

int wc_AesCmacVerify_ex(Cmac* cmac, const byte* check, word32 checkSz,
                        const byte* in, word32 inSz,
                        const byte* key, word32 keySz,
                        void* heap, int devId);

int wc_CmacSetTagLen(Cmac* cmac, word32 tagLen);
```

C.12 coding.h

C.12.1 Functions

	Name
int	**Base64_Decode**(const byte * in, word32 inLen, byte * out, word32 * outLen)This function decodes the given Base64 encoded input, in, and stores the result in the output buffer out. It also sets the size written to the output buffer in the variable outLen.

	Name
int	**Base64_Encode**(const byte * in, word32 inLen, byte * out, word32 * outLen)This function encodes the given input, in, and stores the Base64 encoded result in the output buffer out. It writes the data with the traditional ‘ ’ line endings, instead of escaped %0A line endings. Upon successfully completing, this function also sets outLen to the number of bytes written to the output buffer. If there is enough room in out to store an extra byte, a NULL terminator will be added. This will NOT be included in outLen.
int	**Base64_EncodeEsc**(const byte * in, word32 inLen, byte * out, word32 * outLen)This function encodes the given input, in, and stores the Base64 encoded result in the output buffer out. It writes the data with %0A escaped line endings instead of ‘ ’ line endings. Upon successfully completing, this function also sets outLen to the number of bytes written to the output buffer.
int	**Base64_Encode_NoNl**(const byte * in, word32 inLen, byte * out, word32 * outLen)This function encodes the given input, in, and stores the Base64 encoded result in the output buffer out. It writes the data with no new lines. Upon successfully completing, this function also sets outLen to the number of bytes written to the output buffer.
int	**Base16_Decode**(const byte * in, word32 inLen, byte * out, word32 * outLen)This function decodes the given Base16 encoded input, in, and stores the result in the output buffer out. It also sets the size written to the output buffer in the variable outLen.
int	**Base16_Encode**(const byte * in, word32 inLen, byte * out, word32 * outLen)Encode input to base16 output. If there is enough room in out to store an extra byte, a NULL terminator will be added and included in outLen.
int	**Base64_Decode_nonCT**(const byte * in, word32 inLen, byte * out, word32 * outLen)This function decodes Base64 encoded input without using constant-time operations. This is faster than the constant-time version but may be vulnerable to timing attacks. Use only when timing attacks are not a concern.

	Name
int	**wc_Utf8_DecodeChar**(const byte * in, word32 inLen, word32 * inOutIdx, word32 * cp)This function decodes the UTF-8 encoding of one code point. The encodings that RFC 3629 Section 3 requires a decoder to reject are rejected: overlong forms, the code points reserved for UTF-16 surrogates, code points past the end of the Unicode range, the five and six octet forms that RFC 3629 removed, and sequences whose continuation octets are missing or malformed. Letting any of those decode to a character would allow one octet sequence to impersonate another. The index is only advanced when a code point is decoded, so a caller that wants to continue past a bad sequence chooses for itself how far to skip.

C.12.2 Functions Documentation

C.12.2.1 function Base64_Decode

```
int Base64_Decode(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

This function decodes the given Base64 encoded input, in, and stores the result in the output buffer out. It also sets the size written to the output buffer in the variable outLen.

Parameters:

- **in** pointer to the input buffer to decode
- **inLen** length of the input buffer to decode
- **out** pointer to the output buffer in which to store the decoded message
- **outLen** pointer to the length of the output buffer. Updated with the bytes written at the end of the function call

See:

- Base64_Encode
- Base16_Decode

Return:

- 0 Returned upon successfully decoding the Base64 encoded input
- BAD_FUNC_ARG Returned if the output buffer is too small to store the decoded input
- ASN_INPUT_E Returned if a character in the input buffer falls outside of the Base64 range ([A-Za-z0-9+/=]) or if there is an invalid line ending in the Base64 encoded input

Example

```
byte encoded[] = { // initialize text to decode };
byte decoded[sizeof(encoded)];
// requires at least (sizeof(encoded) * 3 + 3) / 4 room
```

```
int outLen = sizeof(decoded);

if( Base64_Decode(encoded,sizeof(encoded), decoded, &outLen) != 0 ) {
    // error decoding input buffer
}
```

C.12.2.2 function Base64_Encode

```
int Base64_Encode(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

This function encodes the given input, in, and stores the Base64 encoded result in the output buffer out. It writes the data with the traditional ‘ ’ line endings, instead of escaped %0A line endings. Upon successfully completing, this function also sets outLen to the number of bytes written to the output buffer. If there is enough room in out to store an extra byte, a NULL terminator will be added. This will NOT be included in outLen.

Parameters:

- **in** pointer to the input buffer to encode
- **inLen** length of the input buffer to encode
- **out** pointer to the output buffer in which to store the encoded message
- **outLen** pointer to the length of the output buffer in which to store the encoded message

See:

- Base64_EncodeEsc
- Base64_Decode

Return:

- 0 Returned upon successfully decoding the Base64 encoded input
- BAD_FUNC_ARG Returned if the output buffer is too small to store the encoded input
- BUFFER_E Returned if the output buffer runs out of room while encoding

Example

```
byte plain[] = { // initialize text to encode };
byte encoded[MAX_BUFFER_SIZE];

int outLen = sizeof(encoded);

if( Base64_Encode(plain, sizeof(plain), encoded, &outLen) != 0 ) {
    // error encoding input buffer
}
```

C.12.2.3 function Base64_EncodeEsc

```
int Base64_EncodeEsc(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

This function encodes the given input, in, and stores the Base64 encoded result in the output buffer out. It writes the data with %0A escaped line endings instead of ‘ ’ line endings. Upon successfully completing, this function also sets outLen to the number of bytes written to the output buffer.

Parameters:

- **in** pointer to the input buffer to encode
- **inLen** length of the input buffer to encode
- **out** pointer to the output buffer in which to store the encoded message
- **outLen** pointer to the length of the output buffer in which to store the encoded message

See:

- Base64_Encode
- Base64_Decode

Return:

- 0 Returned upon successfully decoding the Base64 encoded input
- BAD_FUNC_ARG Returned if the output buffer is too small to store the encoded input
- BUFFER_E Returned if the output buffer runs out of room while encoding
- ASN_INPUT_E Returned if there is an error processing the decode on the input message

Example

```
byte plain[] = { // initialize text to encode };
byte encoded[MAX_BUFFER_SIZE];

int outLen = sizeof(encoded);

if( Base64_EncodeEsc(plain, sizeof(plain), encoded, &outLen) != 0 ) {
    // error encoding input buffer
}
```

C.12.2.4 function Base64_Encode_NoNl

```
int Base64_Encode_NoNl(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

This function encodes the given input, in, and stores the Base64 encoded result in the output buffer out. It writes the data with no new lines. Upon successfully completing, this function also sets outLen to the number of bytes written to the output buffer.

Parameters:

- **in** pointer to the input buffer to encode
- **inLen** length of the input buffer to encode
- **out** pointer to the output buffer in which to store the encoded message
- **outLen** pointer to the length of the output buffer in which to store the encoded message

See:

- Base64_Encode
- Base64_Decode

Return:

- 0 Returned upon successfully decoding the Base64 encoded input

- BAD_FUNC_ARG Returned if the output buffer is too small to store the encoded input
- BUFFER_E Returned if the output buffer runs out of room while encoding
- ASN_INPUT_E Returned if there is an error processing the decode on the input message

Example

```
byte plain[] = { // initialize text to encode };
byte encoded[MAX_BUFFER_SIZE];
int outLen = sizeof(encoded);
if( Base64_Encode_NoNl(plain, sizeof(plain), encoded, &outLen) != 0 ) {
    // error encoding input buffer
}
```

C.12.2.5 function Base16_Decode

```
int Base16_Decode(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

This function decodes the given Base16 encoded input, in, and stores the result in the output buffer out. It also sets the size written to the output buffer in the variable outLen.

Parameters:

- **in** pointer to the input buffer to decode
- **inLen** length of the input buffer to decode
- **out** pointer to the output buffer in which to store the decoded message
- **outLen** pointer to the length of the output buffer. Updated with the bytes written at the end of the function call

See:

- Base64_Encode
- Base64_Decode
- Base16_Encode

Return:

- 0 Returned upon successfully decoding the Base16 encoded input
- BAD_FUNC_ARG Returned if the output buffer is too small to store the decoded input or if the input length is not a multiple of two
- ASN_INPUT_E Returned if a character in the input buffer falls outside of the Base16 range ([0-9A-F])

Example

```
byte encoded[] = { // initialize text to decode };
byte decoded[sizeof(encoded)];
int outLen = sizeof(decoded);

if( Base16_Decode(encoded,sizeof(encoded), decoded, &outLen) != 0 ) {
    // error decoding input buffer
}
```

C.12.2.6 function Base16_Encode

```
int Base16_Encode(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

Encode input to base16 output. If there is enough room in out to store an extra byte, a NULL terminator will be added and included in outLen.

Parameters:

- **in** Pointer to input buffer to be encoded.
- **inLen** Length of input buffer.
- **out** Pointer to output buffer.
- **outLen** Length of output buffer. Is set to len of encoded output.

See:

- Base64_Encode
- Base64_Decode
- Base16_Decode

Return:

- 0 Success
- BAD_FUNC_ARG Returns if in, out, or outLen is null or if outLen is less than 2 times inLen plus 1.

Example

```
byte in[] = { // Contents of something to be encoded };
byte out[NECESSARY_OUTPUT_SIZE];
word32 outSz = sizeof(out);

if(Base16_Encode(in, sizeof(in), out, &outSz) != 0)
{
    // Handle encode error
}
```

C.12.2.7 function Base64_Decode_nonCT

```
int Base64_Decode_nonCT(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

This function decodes Base64 encoded input without using constant-time operations. This is faster than the constant-time version but may be vulnerable to timing attacks. Use only when timing attacks are not a concern.

Parameters:

- **in** pointer to the input buffer to decode
- **inLen** length of the input buffer to decode
- **out** pointer to the output buffer to store decoded message
- **outLen** pointer to length of output buffer; updated with bytes written

See:

- Base64_Decode
- Base64_Encode

Return:

- 0 On successfully decoding the Base64 encoded input.
- BAD_FUNC_ARG If the output buffer is too small to store the decoded input.
- ASN_INPUT_E If a character in the input buffer falls outside of the Base64 range or if there is an invalid line ending.
- BUFFER_E If running out of buffer while decoding.

Example

```
byte encoded[] = "SGVsbG8gV29ybGQ="; // "Hello World" in Base64
byte decoded[64];
word32 outLen = sizeof(decoded);

int ret = Base64_Decode_nonCT(encoded, sizeof(encoded)-1, decoded,
                              &outLen);
if (ret != 0) {
    // error decoding input
}
// decoded now contains "Hello World"
```

C.12.2.8 function wc_Utf8_DecodeChar

```
int wc_Utf8_DecodeChar(
    const byte * in,
    word32 inLen,
    word32 * inOutIdx,
    word32 * cp
)
```

This function decodes the UTF-8 encoding of one code point. The encodings that RFC 3629 Section 3 requires a decoder to reject are rejected: overlong forms, the code points reserved for UTF-16 surrogates, code points past the end of the Unicode range, the five and six octet forms that RFC 3629 removed, and sequences whose continuation octets are missing or malformed. Letting any of those decode to a character would allow one octet sequence to impersonate another. The index is only advanced when a code point is decoded, so a caller that wants to continue past a bad sequence chooses for itself how far to skip.

Parameters:

- **in** pointer to the buffer holding UTF-8 encoded text
- **inLen** length of the input buffer in octets
- **inOutIdx** pointer to the index of the first octet to decode; updated to the index of the first octet after the code point decoded
- **cp** pointer to store the decoded code point

Return:

- 0 On successfully decoding one code point.
- BAD_FUNC_ARG If in, inOutIdx or cp is NULL.
- BUFFER_E If no octets remain, or if the sequence needs more continuation octets than the buffer holds.
- ASN_INPUT_E If the octets are not a valid encoding of a code point.

Example

```
const byte text[] = { 0xe2, 0x82, 0xac }; // U+20AC
word32 idx = 0;
word32 cp;

while (idx < sizeof(text)) {
    if (wc_Utf8_DecodeChar(text, sizeof(text), &idx, &cp) != 0) {
        // not valid UTF-8 at idx
        break;
    }
    // cp holds the next code point
}
```

C.12.3 Source code

```
int Base64_Decode(const byte* in, word32 inLen, byte* out,
                                 word32* outLen);

int Base64_Encode(const byte* in, word32 inLen, byte* out,
                                    word32* outLen);

int Base64_EncodeEsc(const byte* in, word32 inLen, byte* out,
                                    word32* outLen);

int Base64_Encode_NoNl(const byte* in, word32 inLen, byte* out,
                                    word32* outLen);

int Base16_Decode(const byte* in, word32 inLen, byte* out, word32* outLen);

int Base16_Encode(const byte* in, word32 inLen, byte* out, word32* outLen);

int Base64_Decode_nonCT(const byte* in, word32 inLen, byte* out,
                         word32* outLen);

int wc_Utf8_DecodeChar(const byte* in, word32 inLen, word32* inOutIdx,
                        word32* cp);
```

C.13 compress.h

C.13.1 Functions

	Name
int	**wc_Compress**(byte * out, word32 outSz, const byte * in, word32 inSz, word32 flags)This function compresses the given input data using Huffman coding and stores the output in out. Note that the output buffer should still be larger than the input buffer because there exists a certain input for which there will be no compression possible, which will still require a lookup table. It is recommended that one allocate srcSz + 0.1% + 12 for the output buffer.

	Name
int	**wc_DeCompress**(byte * out, word32 outSz, const byte * in, word32 inSz)This function decompresses the given compressed data using Huffman coding and stores the output in out.
int	**wc_Compress_ex**(byte * out, word32 outSz, const byte * in, word32 inSz, word32 flags, word32 windowBits)This function compresses the given input data using Huffman coding with extended parameters. This is similar to wc_Compress but allows specification of compression flags and window bits for more control over the compression process.
int	**wc_DeCompress_ex**(byte * out, word32 outSz, const byte * in, word32 inSz, int windowBits)This function decompresses the given compressed data using Huffman coding with extended parameters. This is similar to wc_DeCompress but allows specification of window bits for more control over the decompression process.
int	**wc_DeCompressDynamic**(byte ** out, int max, int memoryType, const byte * in, word32 inSz, int windowBits, void * heap)This function decompresses the given compressed data using Huffman coding with dynamic memory allocation. The output buffer is allocated dynamically and the caller is responsible for freeing it.

C.13.2 Functions Documentation

C.13.2.1 function wc_Compress

```
int wc_Compress(
    byte * out,
    word32 outSz,
    const byte * in,
    word32 inSz,
    word32 flags
)
```

This function compresses the given input data using Huffman coding and stores the output in out. Note that the output buffer should still be larger than the input buffer because there exists a certain input for which there will be no compression possible, which will still require a lookup table. It is recommended that one allocate srcSz + 0.1% + 12 for the output buffer.

Parameters:

- **out** pointer to the output buffer in which to store the compressed data
- **outSz** size available in the output buffer for storage
- **in** pointer to the buffer containing the message to compress
- **inSz** size of the input message to compress
- **flags** flags to control how compression operates. Use 0 for normal decompression

See: wc_DeCompress

Return:

- On successfully compressing the input data, returns the number of bytes stored in the output buffer
- COMPRESS_INIT_E Returned if there is an error initializing the stream for compression
- COMPRESS_E Returned if an error occurs during compression

Example

```
byte message[] = { // initialize text to compress };
byte compressed[(sizeof(message) + sizeof(message) * .001 + 12 )];
// Recommends at least srcSz + .1% + 12

if( wc_Compress(compressed, sizeof(compressed), message, sizeof(message),
0) != 0){
    // error compressing data
}
```

C.13.2.2 function wc_DeCompress

```
int wc_DeCompress(
    byte * out,
    word32 outSz,
    const byte * in,
    word32 inSz
)
```

This function decompresses the given compressed data using Huffman coding and stores the output in out.

Parameters:

- **out** pointer to the output buffer in which to store the decompressed data
- **outSz** size available in the output buffer for storage
- **in** pointer to the buffer containing the message to decompress
- **inSz** size of the input message to decompress

See: wc_Compress

Return:

- Success On successfully decompressing the input data, returns the number of bytes stored in the output buffer
- COMPRESS_INIT_E: Returned if there is an error initializing the stream for compression
- COMPRESS_E: Returned if an error occurs during compression

Example

```
byte compressed[] = { // initialize compressed message };
byte decompressed[MAX_MESSAGE_SIZE];

if( wc_DeCompress(decompressed, sizeof(decompressed),
compressed, sizeof(compressed)) != 0 ) {
    // error decompressing data
}
```

C.13.2.3 function wc_Compress_ex

```
int wc_Compress_ex(
    byte * out,
    word32 outSz,
    const byte * in,
    word32 inSz,
    word32 flags,
    word32 windowBits
)
```

This function compresses the given input data using Huffman coding with extended parameters. This is similar to wc_Compress but allows specification of compression flags and window bits for more control over the compression process.

Parameters:

- **out** pointer to the output buffer in which to store the compressed data
- **outSz** size available in the output buffer for storage
- **in** pointer to the buffer containing the message to compress
- **inSz** size of the input message to compress
- **flags** flags to control how compression operates
- **windowBits** the base two logarithm of the window size (8..15)

See:

- wc_Compress
- wc_DeCompress_ex

Return:

- On successfully compressing the input data, returns the number of bytes stored in the output buffer
- COMPRESS_INIT_E Returned if there is an error initializing the stream for compression
- COMPRESS_E Returned if an error occurs during compression

Example

```
byte message[] = { // initialize text to compress };
byte compressed[(sizeof(message) + sizeof(message) * .001 + 12)];
word32 flags = 0;
word32 windowBits = 15; // 32KB window

int ret = wc_Compress_ex(compressed, sizeof(compressed), message,
                         sizeof(message), flags, windowBits);
if (ret < 0) {
    // error compressing data
}
```

C.13.2.4 function wc_DeCompress_ex

```
int wc_DeCompress_ex(
    byte * out,
    word32 outSz,
    const byte * in,
    word32 inSz,
    int windowBits
)
```

This function decompresses the given compressed data using Huffman coding with extended parameters. This is similar to wc_DeCompress but allows specification of window bits for more control over the decompression process.

Parameters:

- **out** pointer to the output buffer in which to store the decompressed data
- **outSz** size available in the output buffer for storage
- **in** pointer to the buffer containing the message to decompress
- **inSz** size of the input message to decompress
- **windowBits** the base two logarithm of the window size (8..15)

See:

- wc_DeCompress
- wc_Compress_ex

Return:

- On successfully decompressing the input data, returns the number of bytes stored in the output buffer
- COMPRESS_INIT_E Returned if there is an error initializing the stream for decompression
- COMPRESS_E Returned if an error occurs during decompression

Example

```
byte compressed[] = { // initialize compressed message };
byte decompressed[MAX_MESSAGE_SIZE];
int windowBits = 15;

int ret = wc_DeCompress_ex(decompressed, sizeof(decompressed),
                           compressed, sizeof(compressed),
                           windowBits);
if (ret < 0) {
    // error decompressing data
}
```

C.13.2.5 function wc_DeCompressDynamic

```
int wc_DeCompressDynamic(
    byte ** out,
    int max,
    int memoryType,
    const byte * in,
    word32 inSz,
    int windowBits,
    void * heap
)
```

This function decompresses the given compressed data using Huffman coding with dynamic memory allocation. The output buffer is allocated dynamically and the caller is responsible for freeing it.

Parameters:

- **out** pointer to pointer that will be set to the allocated output buffer
- **max** maximum size to allocate for output buffer
- **memoryType** type of memory to allocate (DYNAMIC_TYPE_TMP_BUFFER)
- **in** pointer to the buffer containing the message to decompress
- **inSz** size of the input message to decompress
- **windowBits** the base two logarithm of the window size (8..15)

- **heap** heap hint for memory allocation (can be NULL)

See:

- wc_DeCompress
- wc_DeCompress_ex

Return:

- On successfully decompressing the input data, returns the number of bytes stored in the output buffer
- COMPRESS_INIT_E Returned if there is an error initializing the stream for decompression
- COMPRESS_E Returned if an error occurs during decompression
- MEMORY_E Returned if memory allocation fails

Example

```
byte compressed[] = { // initialize compressed message };
byte* decompressed = NULL;
int max = 1024 * 1024; // 1MB max

int ret = wc_DeCompressDynamic(&decompressed, max,
                               DYNAMIC_TYPE_TMP_BUFFER, compressed,
                               sizeof(compressed), 15, NULL);
if (ret < 0) {
    // error decompressing data
}
else {
    // use decompressed data
    XFREE(decompressed, NULL, DYNAMIC_TYPE_TMP_BUFFER);
}
```

C.13.3 Source code

```
int wc_Compress(byte* out, word32 outSz, const byte* in, word32 inSz, word32
 ↪  flags);

int wc_DeCompress(byte* out, word32 outSz, const byte* in, word32 inSz);

int wc_Compress_ex(byte* out, word32 outSz, const byte* in, word32 inSz,
                   word32 flags, word32 windowBits);

int wc_DeCompress_ex(byte* out, word32 outSz, const byte* in, word32 inSz,
                     int windowBits);

int wc_DeCompressDynamic(byte** out, int max, int memoryType,
                         const byte* in, word32 inSz, int windowBits,
                         void* heap);
```

C.14 cryptocb.h

C.14.1 Functions

	Name
int	**wc_CryptoCb_RegisterDevice**(int devId, CryptoDevCallbackFunc cb, void * ctx)This function registers a unique device identifier (devID) and callback function for offloading crypto operations to external hardware such as Key Store, Secure Element, HSM, PKCS11 or TPM.
int	**wc_CryptoCb_IsDeviceRegistered**(int devId)This function reports whether a crypto callback device identifier (devID) is currently registered. It is useful for checking registration state before calling wc_CryptoCb_RegisterDevice, which now rejects an already-registered devID with ALREADY_E.
void	**wc_CryptoCb_UnRegisterDevice**(int devId)This function un_registers a unique device identifier (devID) callback function.
int	**wc_CryptoCb_DefaultDevID**(void)This function returns the default device ID for crypto callbacks. This is useful when you want to get the device ID that was set as the default for the library.
void	**wc_CryptoCb_SetDeviceFindCb**(CryptoDevCallbackFind cb)This function sets a callback for finding crypto devices. The callback is invoked when a device ID needs to be resolved to a device context. This is useful for dynamic device management.
void	**wc_CryptoCb_InfoString**(wc_CryptoInfo * info)This function converts a wc_CryptoInfo structure to a human-readable string for debugging purposes. The string is printed to stdout and describes the cryptographic operation being performed.
int	**wc_CryptoCb_AesSetKey**(Aes * aes, const byte * key, word32 keySz)Import an AES key into a CryptoCB device for hardware offload.
int	**wc_CryptoCb_EccMakePub**(ecc_key * key, ecc_point * pubOut)Offload deriving an ECC public key Q = d*G from its private key to a CryptoCB device.
int	**wc_CryptoCb_EccCheckPubKey**(ecc_key * key, int checkOrder, int checkPriv)Offload validating an ECC key to a CryptoCB device.
int	**wc_CryptoCb_Ed25519MakePub**(ed25519_key * key, byte * pubKey, word32 pubKeySz)Offload deriving an Ed25519 public key from its private key to a CryptoCB device.
int	**wc_CryptoCb_Ed25519CheckKey**(ed25519_key * key)Offload validating an Ed25519 key to a CryptoCB device.

	Name
int	**wc_CryptoCb_Ed448MakePub**(ed448_key * key, byte * pubKey, word32 pubKeySz)Offload deriving an Ed448 public key from its private key to a CryptoCB device.
int	**wc_CryptoCb_Ed448CheckKey**(ed448_key * key)Offload validating an Ed448 key to a CryptoCB device.

C.14.2 Functions Documentation

C.14.2.1 function wc_CryptoCb_RegisterDevice

```
int wc_CryptoCb_RegisterDevice(
    int devId,
    CryptoDevCallbackFunc cb,
    void * ctx
)
```

This function registers a unique device identifier (devID) and callback function for offloading crypto operations to external hardware such as Key Store, Secure Element, HSM, PKCS11 or TPM.

Parameters:

- **devId** any unique value, not -2 (INVALID_DEVID)
- **cb** a callback function with prototype: typedef int (*CryptoDevCallbackFunc)(int devId, wc_CryptoInfo* info, void* ctx);
- **ctx** user context handed back to the callback on every invocation. With WOLF_CRYPTO_CB_CMD, a register command handler may replace it by updating info->cmd.ctx on success.

See:

- wc_CryptoCb_UnRegisterDevice
- wolfSSL_SetDevId
- wolfSSL_CTX_SetDevId

Return:

- CRYPTOCB_UNAVAILABLE to fallback to using software crypto
- 0 for success
- ALREADY_E if devId is already registered. A devId must be un-registered with wc_CryptoCb_UnRegisterDevice before it can be registered again; re-registering an active devId is not an in-place update.
- BAD_FUNC_ARG if devId is INVALID_DEVID (-2)
- negative value for failure

For STSAFE with Crypto Callbacks example see wolfcrypt/src/port/st/stsafe.c and the wolfSSL_STSAFE_CryptoDevCb function.

For TPM based crypto callbacks example see the wolfTPM2_CryptoDevCb function in wolfTPM src/tpm2_wrap.c

With WOLF_CRYPTO_CB_CMD defined, the callback is invoked during this call with a WC_CRYPTOCB_CMD_TYPE_REGIST command (and later with WC_CRYPTOCB_CMD_TYPE_UNREGISTER from wc_CryptoCb_UnRegisterDevice). A register command handler may itself call wc_CryptoCb_RegisterDevice to register additional devIds, such as a driver exposing several devices. It must not dispatch through, or re-register, the devId it is being registered with: that devId is not published until this call returns, so a nested registration of it is not caught by the ALREADY_E check and consumes fresh slots until the table fills and BUFFER_E unwinds them. If the handler returns an error other than CRYPTOCB_UNAVAILABLE or

NOT_COMPILED_IN, the registration is unwound and the error returned; devIds the handler already registered stay registered and are the caller's to clean up.

Example

```
#include <wolfssl/wolfcrypt/settings.h>
#include <wolfssl/wolfcrypt/cryptocb.h>
static int myCryptoCb_Func(int devId, wc_CryptoInfo* info, void* ctx)
{
    int ret = CRYPTOCB_UNAVAILABLE;

    if (info->algo_type == WC_ALGO_TYPE_PK) {
    #ifndef NO_RSA
        if (info->pk.type == WC_PK_TYPE_RSA) {
            switch (info->pk.rsa.type) {
                case RSA_PUBLIC_ENCRYPT:
                case RSA_PUBLIC_DECRYPT:
                    // RSA public op
                    ret = wc_RsaFunction(
                        info->pk.rsa.in, info->pk.rsa.inLen,
                        info->pk.rsa.out, info->pk.rsa.outLen,
                        info->pk.rsa.type, info->pk.rsa.key,
                        info->pk.rsa.rng);
                    break;
                case RSA_PRIVATE_ENCRYPT:
                case RSA_PRIVATE_DECRYPT:
                    // RSA private op
                    ret = wc_RsaFunction(
                        info->pk.rsa.in, info->pk.rsa.inLen,
                        info->pk.rsa.out, info->pk.rsa.outLen,
                        info->pk.rsa.type, info->pk.rsa.key,
                        info->pk.rsa.rng);
                    break;
            }
        }
    #endif
    #if defined(WC_RSA_PSS) && !defined(NO_RSA)
        if (info->pk.type == WC_PK_TYPE_RSA_PSS) {
            // RSA-PSS sign/verify
            ret = wc_RsaPSS_Sign_ex(
                info->pk.rsa.in, info->pk.rsa.inLen,
                info->pk.rsa.out, *info->pk.rsa.outLen,
                WC_HASH_TYPE_SHA256, WC_MGF1SHA256,
                RSA_PSS_SALT_LEN_DEFAULT,
                info->pk.rsa.key, info->pk.rsa.rng);
        }
    #endif
    #ifdef HAVE_ECC
        if (info->pk.type == WC_PK_TYPE_ECDSA_SIGN) {
            // ECDSA
            ret = wc_ecc_sign_hash(
                info->pk.eccsign.in, info->pk.eccsign.inlen,
                info->pk.eccsign.out, info->pk.eccsign.outlen,
                info->pk.eccsign.rng, info->pk.eccsign.key);
        }
```

```
    #endif
    #ifdef HAVE_ED25519
        if (info->pk.type == WC_PK_TYPE_ED25519_SIGN) {
            // ED25519 sign
            ret = wc_ed25519_sign_msg_ex(
                info->pk.ed25519sign.in, info->pk.ed25519sign.inLen,
                info->pk.ed25519sign.out, info->pk.ed25519sign.outLen,
                info->pk.ed25519sign.key, info->pk.ed25519sign.type,
                info->pk.ed25519sign.context,
                info->pk.ed25519sign.contextLen);
        }
    #endif
    }
    return ret;
}

int devId = 1;
wc_CryptoCb_RegisterDevice(devId, myCryptoCb_Func, &myCtx);
wolfSSL_CTX_SetDevId(ctx, devId);
```

C.14.2.2 function wc_CryptoCb_IsDeviceRegistered

```
int wc_CryptoCb_IsDeviceRegistered(
    int devId
)
```

This function reports whether a crypto callback device identifier (devID) is currently registered. It is useful for checking registration state before calling wc_CryptoCb_RegisterDevice, which now rejects an already-registered devID with ALREADY_E.

Parameters:

- **devId** the device identifier to query

See:

- wc_CryptoCb_RegisterDevice
- wc_CryptoCb_UnRegisterDevice

Return:

- 1 if the device ID is registered
- 0 if the device ID is not registered, or if devId is INVALID_DEVID (-2)

Example

```
if (!wc_CryptoCb_IsDeviceRegistered(devId)) {
    wc_CryptoCb_RegisterDevice(devId, myCryptoCb_Func, &myCtx);
}
```

C.14.2.3 function wc_CryptoCb_UnRegisterDevice

```
void wc_CryptoCb_UnRegisterDevice(
    int devId
)
```

This function un-registers a unique device identifier (devID) callback function.

Parameters:

- **devId** any unique value, not -2 (INVALID_DEVID)

See:

- wc_CryptoCb_RegisterDevice
- wolfSSL_SetDevId
- wolfSSL_CTX_SetDevId

Return: none No returns.

Example

```
wc_CryptoCb_UnRegisterDevice(devId);
devId = INVALID_DEVID;
wolfSSL_CTX_SetDevId(ctx, devId);
```

C.14.2.4 function wc_CryptoCb_DefaultDevID

```
int wc_CryptoCb_DefaultDevID(
    void
)
```

This function returns the default device ID for crypto callbacks. This is useful when you want to get the device ID that was set as the default for the library.

See:

- wc_CryptoCb_RegisterDevice
- wc_CryptoCb_UnRegisterDevice

Return: The default device ID, or INVALID_DEVID if no default is set.

Example

```
int devId = wc_CryptoCb_DefaultDevID();
if (devId != INVALID_DEVID) {
    // default device ID is set
}
```

C.14.2.5 function wc_CryptoCb_SetDeviceFindCb

```
void wc_CryptoCb_SetDeviceFindCb(
    CryptoDevCallbackFind cb
)
```

This function sets a callback for finding crypto devices. The callback is invoked when a device ID needs to be resolved to a device context. This is useful for dynamic device management.

Parameters:

- **cb** callback function with prototype: typedef void* (*CryptoDevCallbackFind)(int devId);

See: wc_CryptoCb_RegisterDevice

Return: none No returns.

Example

```
void* myDeviceFindCb(int devId) {
    // lookup device context by ID
    return deviceContext;
}
```

```
wc_CryptoCb_SetDeviceFindCb(myDeviceFindCb);
```

C.14.2.6 function wc_CryptoCb_InfoString

```
void wc_CryptoCb_InfoString(
    wc_CryptoInfo * info
)
```

This function converts a wc_CryptoInfo structure to a human-readable string for debugging purposes. The string is printed to stdout and describes the cryptographic operation being performed.

Parameters:

- **info** pointer to the wc_CryptoInfo structure to convert

See: wc_CryptoCb_RegisterDevice

Return: none No returns.

Example

```
int myCryptoCb(int devId, wc_CryptoInfo* info, void* ctx) {
    // print debug info about the operation
    wc_CryptoCb_InfoString(info);

    // handle the operation
    return CRYPTOCB_UNAVAILABLE;
}
```

C.14.2.7 function wc_CryptoCb_AesSetKey

```
int wc_CryptoCb_AesSetKey(
    Aes * aes,
    const byte * key,
    word32 keySz
)
```

Import an AES key into a CryptoCB device for hardware offload.

Parameters:

- **aes** AES context
- **key** Pointer to raw AES key material
- **keySz** Size of key in bytes

See:

- wc_CryptoCb_RegisterDevice
- wc_AesInit

Return:

- 0 on success
- CRYPTOCB_UNAVAILABLE if device does not support this operation
- BAD_FUNC_ARG on invalid parameters

This function allows AES keys to be handled by an external device (e.g. Secure Element or HSM). When supported, the device callback stores the key internally and sets an opaque handle in aes->devCtx.

When CryptoCB AES SetKey support is enabled (WOLF_CRYPTO_CB_AES_SETKEY), wolfCrypt routes AES-GCM operations through the CryptoCB interface.

TLS Builds (Default):**

- Key bytes ARE stored in wolfCrypt memory (devKey) for fallback
- GCM tables ARE generated for software fallback
- Provides hardware acceleration with automatic fallback Crypto-Only Builds (–disable-tls):**
- Key bytes NOT stored in wolfCrypt memory (true key isolation)
- GCM tables skipped (true hardware offload)
- Callback must handle all GCM operations (SetKey, Encrypt, Decrypt, Free)

If the callback returns success (0), full AES-GCM offload is assumed. The callback must handle SetKey, Encrypt, Decrypt, and Free operations.

Example

```
#include <wolfssl/wolfcrypt/cryptocb.h>
#include <wolfssl/wolfcrypt/aes.h>

Aes aes;
byte key[32] = { /* 256-bit key */ };
int devId = 1;

/* Register your CryptoCB callback first */
wc_CryptoCb_RegisterDevice(devId, myCryptoCallback, NULL);

wc_AesInit(&aes, NULL, devId);
/* wc_AesGcmSetKey internally calls wc_CryptoCb_AesSetKey */
if (wc_CryptoCb_AesSetKey(&aes, key, sizeof(key)) == 0) {
    /* Key successfully imported to device via callback */
    /* aes.devCtx now contains device handle */
    /* Full GCM offload is assumed - callback must handle all operations */
}
```

C.14.2.8 function wc_CryptoCb_EccMakePub

```
int wc_CryptoCb_EccMakePub(
    ecc_key * key,
    ecc_point * pubOut
)
```

Offload deriving an ECC public key Q = d*G from its private key to a CryptoCB device.

Parameters:

- **key** ECC key providing the device id, curve identity, heap hint and the private scalar
- **pubOut** [out] resulting affine public point Q = d*G

See:

- wc_CryptoCb_RegisterDevice
- wc_ecc_make_pub

Return:

- 0 on success
- CRYPTOCB_UNAVAILABLE if no device handles the operation, or key or pubOut is NULL (wolfCrypt falls back to software, which reports the argument error)

Used by wc_ecc_make_pub / wc_ecc_make_pub_ex. The callback boundary is math-free: the resulting public point crosses as X9.63 uncompressed bytes (0x04 || X || Y, each ordinate zero-padded to the curve size) in wc_CryptoInfo.pk.ecc_make_pub (pubOut / pubOutSz). This wrapper performs all

bignum (de)serialization, so a device handler only deals with byte arrays and never with wolfCrypt's internal mp_int representation. The private scalar is taken from the ecc_key (resident in a secure element, or key->k); curve identity comes from key->dp.

C.14.2.9 function wc_CryptoCb_EccCheckPubKey

```
int wc_CryptoCb_EccCheckPubKey(
    ecc_key * key,
    int checkOrder,
    int checkPriv
)
```

Offload validating an ECC key to a CryptoCB device.

Parameters:

- **key** ECC key to validate (curve identity from key->dp)
- **checkOrder** when 1 the caller requested validation that the point has the curve order (point * order == infinity)
- **checkPriv** when 1 the caller also requested validation of the private part (scalar range, consistency with the public point)

See:

- wc_CryptoCb_RegisterDevice
- wc_ecc_check_key

Return:

- 0 if the key is valid
- CRYPTOCB_UNAVAILABLE if no device handles the operation (wolfCrypt falls back to software)

Used by wc_ecc_check_key and the key generation / import validation paths. The public point crosses the (math-free) callback boundary as X9.63 uncompressed bytes in wc_CryptoInfo.pk.ecc_check_pub (pubKey / pubKeySz); this wrapper serializes key->pubkey so a device handler only deals with byte arrays. When the key state is ECC_PRIVATEKEY_ONLY, pubKey is NULL and pubKeySz is 0 so the handler can distinguish "no public point" from an invalid zero-coordinate public point that must be validated and rejected. The caller's intent crosses in checkOrder and checkPriv.

C.14.2.10 function wc_CryptoCb_Ed25519MakePub

```
int wc_CryptoCb_Ed25519MakePub(
    ed25519_key * key,
    byte * pubKey,
    word32 pubKeySz
)
```

Offload deriving an Ed25519 public key from its private key to a CryptoCB device.

Parameters:

- **key** Ed25519 key providing the device id, heap hint and the private key
- **pubKey** [out] resulting compressed public key
- **pubKeySz** size of pubKey buffer, must be ED25519_PUB_KEY_SIZE

See:

- wc_CryptoCb_RegisterDevice
- wc_ed25519_make_public

Return:

- 0 on success
- CRYPTOCB_UNAVAILABLE if no device handles the operation, or key or pubKey is NULL or pubKeySz is wrong (wolfCrypt falls back to software, which reports the argument error)

Used by wc_ed25519_make_public (and so by key generation and private-key import). The boundary is byte-oriented: the device writes the compressed public key into wc_CryptoInfo.pk.ed25519makepub (pub0ut / pub0utSz, always ED25519_PUB_KEY_SIZE). The private key is taken from the ed25519_key (resident in a secure element, or key->k). On success the caller marks the public key set.

C.14.2.11 function wc_CryptoCb_Ed25519CheckKey

```
int wc_CryptoCb_Ed25519CheckKey(
    ed25519_key * key
)
```

Offload validating an Ed25519 key to a CryptoCB device.

Parameters:

- **key** Ed25519 key to validate

See:

- wc_CryptoCb_RegisterDevice
- wc_ed25519_check_key

Return:

- 0 if the key is valid
- CRYPTOCB_UNAVAILABLE if no device handles the operation (wolfCrypt falls back to software)

Used by wc_ed25519_check_key and the key import validation paths. The public key crosses the callback boundary as its compressed wire bytes in wc_CryptoInfo.pk.ed25519checkkey (pubKey / pubKeySz), so a device handler only deals with byte arrays. The dispatch only runs when a public key is present; checkPriv is 1 when a private key is also set or referenced by id or label, and the device should additionally validate priv/pub consistency.

C.14.2.12 function wc_CryptoCb_Ed448MakePub

```
int wc_CryptoCb_Ed448MakePub(
    ed448_key * key,
    byte * pubKey,
    word32 pubKeySz
)
```

Offload deriving an Ed448 public key from its private key to a CryptoCB device.

Parameters:

- **key** Ed448 key providing the device id, heap hint and the private key
- **pubKey** [out] resulting compressed public key
- **pubKeySz** size of pubKey buffer, must be ED448_PUB_KEY_SIZE

See:

- wc_CryptoCb_RegisterDevice
- wc_ed448_make_public

Return:

- 0 on success

- CRYPTOCB_UNAVAILABLE if no device handles the operation, or key or pubKey is NULL or pubKeySz is wrong (wolfCrypt falls back to software, which reports the argument error)

Used by wc_ed448_make_public (and so by key generation and private-key import). The device writes the compressed public key into wc_CryptoInfo.pk.ed448makepub (pub0ut / pub0utSz, always ED448_PUB_KEY_SIZE).

C.14.2.13 function wc_CryptoCb_Ed448CheckKey

```
int wc_CryptoCb_Ed448CheckKey(
    ed448_key * key
)
```

Offload validating an Ed448 key to a CryptoCB device.

Parameters:

- **key** Ed448 key to validate

See:

- wc_CryptoCb_RegisterDevice
- wc_ed448_check_key

Return:

- 0 if the key is valid
- CRYPTOCB_UNAVAILABLE if no device handles the operation (wolfCrypt falls back to software)

Used by wc_ed448_check_key and when checking a device private key against a certificate. The compressed public key is passed in wc_CryptoInfo.pk.ed448checkkey (pubKey / pubKeySz). checkPriv is 1 when a private key is set or referenced by id or label, and the device should also check that it matches the public key.

C.14.3 Source code

```
int  wc_CryptoCb_RegisterDevice(int devId, CryptoDevCallbackFunc cb, void*
 ↪  ctx);

int wc_CryptoCb_IsDeviceRegistered(int devId);

void wc_CryptoCb_UnRegisterDevice(int devId);

int wc_CryptoCb_DefaultDevID(void);

void wc_CryptoCb_SetDeviceFindCb(CryptoDevCallbackFind cb);

void wc_CryptoCb_InfoString(wc_CryptoInfo* info);
 };
    int devId = 1;

    /* Register your CryptoCB callback first */
    wc_CryptoCb_RegisterDevice(devId, myCryptoCallback, NULL);

    wc_AesInit(&aes, NULL, devId);
    /* wc_AesGcmSetKey internally calls wc_CryptoCb_AesSetKey */
    if (wc_CryptoCb_AesSetKey(&aes, key, sizeof(key)) == 0) {
        /* Key successfully imported to device via callback */
```

```
        /* aes.devCtx now contains device handle */
        /* Full GCM offload is assumed - callback must handle all operations */
    }
    \endcode

    \sa wc_CryptoCb_RegisterDevice
    \sa wc_AesInit
*/
int wc_CryptoCb_AesSetKey(Aes* aes, const byte* key, word32 keySz);

int wc_CryptoCb_EccMakePub(ecc_key* key, ecc_point* pubOut);

int wc_CryptoCb_EccCheckPubKey(ecc_key* key, int checkOrder, int checkPriv);

int wc_CryptoCb_Ed25519MakePub(ed25519_key* key, byte* pubKey,
    word32 pubKeySz);

int wc_CryptoCb_Ed25519CheckKey(ed25519_key* key);

int wc_CryptoCb_Ed448MakePub(ed448_key* key, byte* pubKey, word32 pubKeySz);

int wc_CryptoCb_Ed448CheckKey(ed448_key* key);
```

C.15 curve25519.h

C.15.1 Functions

	Name
int	**wc_curve25519_make_key**(WC_RNG * rng, int keysize, curve25519_key * key)This function generates a Curve25519 key using the given random number generator, rng, of the size given (keysize), and stores it in the given curve25519_key structure. It should be called after the key structure has been initialized through wc_curve25519_init().
int	**wc_curve25519_shared_secret**(curve25519_key * private_key, curve25519_key * public_key, byte * out, word32 * outlen)This function computes a shared secret key given a secret private key and a received public key. It stores the generated secret key in the buffer out and assigns the length of the secret key to outlen. Only supports big endian.
int	**wc_curve25519_shared_secret_ex**(curve25519_key * private_key, curve25519_key * public_key, byte * out, word32 * outlen, int endian)This function computes a shared secret key given a secret private key and a received public key. It stores the generated secret key in the buffer out and assigns the length of the secret key to outlen. Supports both big and little endian.

	Name
int	**wc_curve25519_init**(curve25519_key * key)This function initializes a Curve25519 key. It should be called before generating a key for the structure.
void	**wc_curve25519_free**(curve25519_key * key)This function frees a Curve25519 object.
int	**wc_curve25519_import_private**(const byte * priv, word32 privSz, curve25519_key * key)This function imports a curve25519 private key only. (Big endian).
int	**wc_curve25519_import_private_ex**(const byte * priv, word32 privSz, curve25519_key * key, int endian)curve25519 private key import only. (Big or Little endian).
int	**wc_curve25519_import_private_raw**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, curve25519_key * key)This function imports a public-private key pair into a curve25519_key structure. Big endian only.
int	**wc_curve25519_import_private_raw_ex**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, curve25519_key * key, int endian)This function imports a public-private key pair into a curve25519_key structure. Supports both big and little endian.
int	**wc_curve25519_export_private_raw**(curve25519_key * key, byte * out, word32 * outLen)This function exports a private key from a curve25519_key structure and stores it in the given out buffer. It also sets outLen to be the size of the exported key. Big Endian only.
int	**wc_curve25519_export_private_raw_ex**(curve25519_key * key, byte * out, word32 * outLen, int endian)This function exports a private key from a curve25519_key structure and stores it in the given out buffer. It also sets outLen to be the size of the exported key. Can specify whether it's big or little endian.
int	**wc_curve25519_import_public**(const byte * in, word32 inLen, curve25519_key * key)This function imports a public key from the given in buffer and stores it in the curve25519_key structure.
int	**wc_curve25519_import_public_ex**(const byte * in, word32 inLen, curve25519_key * key, int endian)This function imports a public key from the given in buffer and stores it in the curve25519_key structure.

	Name
int	**wc_curve25519_check_public**(const byte * pub, word32 pubSz, int endian)This function checks that a public key buffer holds a valid Curve25519 key value given the endian ordering.
int	**wc_curve25519_export_public**(curve25519_key * key, byte * out, word32 * outLen)This function exports a public key from the given key structure and stores the result in the out buffer. Big endian only.
int	**wc_curve25519_export_public_ex**(curve25519_key * key, byte * out, word32 * outLen, int endian)This function exports a public key from the given key structure and stores the result in the out buffer. Supports both big and little endian.
int	**wc_curve25519_export_key_raw**(curve25519_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz)Export Curve25519 key pair. Big endian only.
int	**wc_curve25519_export_key_raw_ex**(curve25519_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz, int endian)Export curve25519 key pair. Big or little endian.
int	**wc_curve25519_size**(curve25519_key * key)This function returns the key size of the given key structure.
int	**wc_curve25519_make_pub**(int public_size, byte * pub, int private_size, const byte * priv)This function generates a Curve25519 public key from a given private key. This is a lower-level function that operates directly on byte buffers rather than curve25519_key structures.
int	**wc_curve25519_make_pub_blind**(int public_size, byte * pub, int private_size, const byte * priv, WC_RNG * rng)This function generates a Curve25519 public key from a given private key with blinding to resist side-channel attacks. This adds randomization to the scalar multiplication operation.
int	**wc_curve25519_generic**(int public_size, byte * pub, int private_size, const byte * priv, int basepoint_size, const byte * basepoint)This function performs a generic Curve25519 scalar multiplication with a custom basepoint. This allows computing scalar * basepoint for any basepoint, not just the standard generator.

	Name
int	**wc_curve25519_generic_blind**(int public_size, byte * pub, int private_size, const byte * priv, int basepoint_size, const byte * basepoint, WC_RNG * rng)This function performs a generic Curve25519 scalar multiplication with a custom basepoint and blinding to resist side-channel attacks.
int	**wc_curve25519_make_priv**(WC_RNG * rng, int keysize, byte * priv)This function generates a Curve25519 private key using the given random number generator. This is a lower-level function that generates only the private key bytes.
int	**wc_curve25519_init_ex**(curve25519_key * key, void * heap, int devId)This function initializes a Curve25519 key with extended parameters, allowing specification of custom heap and device ID for hardware acceleration.
int	**wc_curve25519_set_rng**(curve25519_key * key, WC_RNG * rng)This function sets the RNG to be used with a Curve25519 key. This is useful for operations that require randomness such as blinded scalar multiplication.
curve25519_key *	**wc_curve25519_new**(void * heap, int devId, int * result_code)This function allocates and initializes a new Curve25519 key structure with extended parameters. The caller is responsible for freeing the key with wc_curve25519_delete. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.
int	**wc_curve25519_delete**(curve25519_key * key, curve25519_key ** key_p)This function frees a Curve25519 key structure that was allocated with wc_curve25519_new and sets the pointer to NULL. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.

C.15.2 Functions Documentation

C.15.2.1 function wc_curve25519_make_key

```
int wc_curve25519_make_key(
    WC_RNG * rng,
    int keysize,
    curve25519_key * key
)
```

This function generates a Curve25519 key using the given random number generator, rng, of the size given (keysize), and stores it in the given curve25519_key structure. It should be called after the key

structure has been initialized through wc_curve25519_init().

Parameters:

- **rng** Pointer to the RNG object used to generate the ecc key.
- **keysize** Size of the key to generate. Must be 32 bytes for curve25519.
- **key** Pointer to the curve25519_key structure in which to store the generated key.

See: wc_curve25519_init

Return:

- 0 Returned on successfully generating the key and and storing it in the given curve25519_key structure.
- ECC_BAD_ARG_E Returned if the input keysize does not correspond to the keysize for a curve25519 key (32 bytes).
- RNG_FAILURE_E Returned if the rng internal status is not DRBG_OK or if there is in generating the next random block with rng.
- BAD_FUNC_ARG Returned if any of the input parameters passed in are NULL.

Example

```
int ret;

curve25519_key key;
wc_curve25519_init(&key); // initialize key
WC_RNG rng;
wc_InitRng(&rng); // initialize random number generator

ret = wc_curve25519_make_key(&rng, 32, &key);
if (ret != 0) {
    // error making Curve25519 key
}
```

C.15.2.2 function wc_curve25519_shared_secret

```
int wc_curve25519_shared_secret(
    curve25519_key * private_key,
    curve25519_key * public_key,
    byte * out,
    word32 * outlen
)
```

This function computes a shared secret key given a secret private key and a received public key. It stores the generated secret key in the buffer out and assigns the length of the secret key to outlen. Only supports big endian.

Parameters:

- **private_key** Pointer to the curve25519_key structure initialized with the user's private key.
- **public_key** Pointer to the curve25519_key structure containing the received public key.
- **out** Pointer to a buffer in which to store the 32 byte computed secret key.
- **outlen** Pointer in which to store the length written to the output buffer.

See:

- wc_curve25519_init
- wc_curve25519_make_key
- wc_curve25519_shared_secret_ex

Return:

- 0 Returned on successfully computing a shared secret key.
- BAD_FUNC_ARG Returned if any of the input parameters passed in are NULL.
- ECC_BAD_ARG_E Returned if the first bit of the public key is set, to avoid implementation fingerprinting.

Example

```
int ret;

byte sharedKey[32];
word32 keySz;
curve25519_key privKey, pubKey;
// initialize both keys

ret = wc_curve25519_shared_secret(&privKey, &pubKey, sharedKey, &keySz);
if (ret != 0) {
    // error generating shared key
}
```

C.15.2.3 function wc_curve25519_shared_secret_ex

```
int wc_curve25519_shared_secret_ex(
    curve25519_key * private_key,
    curve25519_key * public_key,
    byte * out,
    word32 * outlen,
    int endian
)
```

This function computes a shared secret key given a secret private key and a received public key. It stores the generated secret key in the buffer out and assigns the length of the secret key to outlen. Supports both big and little endian.

Parameters:

- **private_key** Pointer to the curve25519_key structure initialized with the user's private key.
- **public_key** Pointer to the curve25519_key structure containing the received public key.
- **out** Pointer to a buffer in which to store the 32 byte computed secret key.
- **outlen** Pointer in which to store the length written to the output buffer.
- **endian** EC25519_BIG_ENDIAN or EC25519_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve25519_init
- wc_curve25519_make_key
- wc_curve25519_shared_secret

Return:

- 0 Returned on successfully computing a shared secret key.
- BAD_FUNC_ARG Returned if any of the input parameters passed in are NULL.
- ECC_BAD_ARG_E Returned if the first bit of the public key is set, to avoid implementation fingerprinting.

Example

```
int ret;

byte sharedKey[32];
word32 keySz;
```

```
curve25519_key privKey, pubKey;
// initialize both keys

ret = wc_curve25519_shared_secret_ex(&privKey, &pubKey, sharedKey, &keySz,
        EC25519_BIG_ENDIAN);
if (ret != 0) {
    // error generating shared key
}
```

C.15.2.4 function wc_curve25519_init

```
int wc_curve25519_init(
    curve25519_key * key
)
```

This function initializes a Curve25519 key. It should be called before generating a key for the structure.

Parameters:

- **key** Pointer to the curve25519_key structure to initialize.

See: wc_curve25519_make_key

Return:

- 0 Returned on successfully initializing the curve25519_key structure.
- BAD_FUNC_ARG Returned when key is NULL.

Example

```
curve25519_key key;
wc_curve25519_init(&key); // initialize key
// make key and proceed to encryption
```

C.15.2.5 function wc_curve25519_free

```
void wc_curve25519_free(
    curve25519_key * key
)
```

This function frees a Curve25519 object.

Parameters:

- **key** Pointer to the key object to free.

See:

- wc_curve25519_init
- wc_curve25519_make_key

Example

```
curve25519_key privKey;
// initialize key, use it to generate shared secret key
wc_curve25519_free(&privKey);
```

C.15.2.6 function wc_curve25519_import_private

```
int wc_curve25519_import_private(
    const byte * priv,
    word32 privSz,
    curve25519_key * key
)
```

This function imports a curve25519 private key only. (Big endian).

Parameters:

- **priv** Pointer to a buffer containing the private key to import.
- **privSz** Length of the private key to import.
- **key** Pointer to the structure in which to store the imported key.

See:

- wc_curve25519_import_private_ex
- wc_curve25519_size

Return:

- 0 Returned on successfully importing private key.
- BAD_FUNC_ARG Returns if key or priv is null.
- ECC_BAD_ARG_E Returns if privSz is not equal to CURVE25519_KEY_SIZE.

Example

```
int ret;

byte priv[] = { Contents of private key };
curve25519_key key;
wc_curve25519_init(&key);

ret = wc_curve25519_import_private(priv, sizeof(priv), &key);
if (ret != 0) {
    // error importing keys
}
```

C.15.2.7 function wc_curve25519_import_private_ex

```
int wc_curve25519_import_private_ex(
    const byte * priv,
    word32 privSz,
    curve25519_key * key,
    int endian
)
```

curve25519 private key import only. (Big or Little endian).

Parameters:

- **priv** Pointer to a buffer containing the private key to import.
- **privSz** Length of the private key to import.
- **key** Pointer to the structure in which to store the imported key.
- **endian** EC25519_BIG_ENDIAN or EC25519_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve25519_import_private
- wc_curve25519_size

Return:

- 0 Returned on successfully importing private key.
- BAD_FUNC_ARG Returns if key or priv is null.
- ECC_BAD_ARG_E Returns if privSz is not equal to CURVE25519_KEY_SIZE.

Example

```
int ret;

byte priv[] = { // Contents of private key };
curve25519_key key;
wc_curve25519_init(&key);

ret = wc_curve25519_import_private_ex(priv, sizeof(priv), &key,
        EC25519_BIG_ENDIAN);
if (ret != 0) {
    // error importing keys
}
```

C.15.2.8 function wc_curve25519_import_private_raw

```
int wc_curve25519_import_private_raw(
    const byte * priv,
    word32 privSz,
    const byte * pub,
    word32 pubSz,
    curve25519_key * key
)
```

This function imports a public-private key pair into a curve25519_key structure. Big endian only.

Parameters:

- **priv** Pointer to a buffer containing the private key to import.
- **privSz** Length of the private key to import.
- **pub** Pointer to a buffer containing the public key to import.
- **pubSz** Length of the public key to import.
- **key** Pointer to the structure in which to store the imported keys.

See:

- wc_curve25519_init
- wc_curve25519_make_key
- wc_curve25519_import_public
- wc_curve25519_export_private_raw

Return:

- 0 Returned on importing into the curve25519_key structure
- BAD_FUNC_ARG Returns if any of the input parameters are null.
- ECC_BAD_ARG_E Returned if the input key's key size does not match the public or private key sizes.

Example

```
int ret;

byte priv[32];
byte pub[32];
```

```
// initialize with public and private keys
curve25519_key key;

wc_curve25519_init(&key);
// initialize key

ret = wc_curve25519_import_private_raw(&priv, sizeof(priv), pub,
        sizeof(pub), &key);
if (ret != 0) {
    // error importing keys
}
```

C.15.2.9 function wc_curve25519_import_private_raw_ex

```
int wc_curve25519_import_private_raw_ex(
    const byte * priv,
    word32 privSz,
    const byte * pub,
    word32 pubSz,
    curve25519_key * key,
    int endian
)
```

This function imports a public-private key pair into a curve25519_key structure. Supports both big and little endian.

Parameters:

- **priv** Pointer to a buffer containing the private key to import.
- **privSz** Length of the private key to import.
- **pub** Pointer to a buffer containing the public key to import.
- **pubSz** Length of the public key to import.
- **key** Pointer to the structure in which to store the imported keys.
- **endian** EC25519_BIG_ENDIAN or EC25519_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve25519_init
- wc_curve25519_make_key
- wc_curve25519_import_public
- wc_curve25519_export_private_raw
- wc_curve25519_import_private_raw

Return:

- 0 Returned on importing into the curve25519_key structure
- BAD_FUNC_ARG Returns if any of the input parameters are null.
- ECC_BAD_ARG_E Returned if or the input key's key size does not match the public or private key sizes

Example

```
int ret;
byte priv[32];
byte pub[32];
// initialize with public and private keys
curve25519_key key;

wc_curve25519_init(&key);
```

```
// initialize key

ret = wc_curve25519_import_private_raw_ex(&priv, sizeof(priv), pub,
        sizeof(pub), &key, EC25519_BIG_ENDIAN);
if (ret != 0) {
    // error importing keys
}
```

C.15.2.10 function wc_curve25519_export_private_raw

```
int wc_curve25519_export_private_raw(
    curve25519_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports a private key from a curve25519_key structure and stores it in the given out buffer. It also sets outLen to be the size of the exported key. Big Endian only.

Parameters:

- **key** Pointer to the structure from which to export the key.
- **out** Pointer to the buffer in which to store the exported key.
- **outLen** On in, is the size of the out in bytes. On out, will store the bytes written to the output buffer.

See:

- wc_curve25519_init
- wc_curve25519_make_key
- wc_curve25519_import_private_raw
- wc_curve25519_export_private_raw_ex

Return:

- 0 Returned on successfully exporting the private key from the curve25519_key structure.
- BAD_FUNC_ARG Returned if any input parameters are NULL.
- ECC_BAD_ARG_E Returned if *outLen is less than wc_curve25519_size().

Example

```
int ret;
byte priv[32];
word32 privSz;

curve25519_key key;
// initialize and make key

ret = wc_curve25519_export_private_raw(&key, priv, &privSz);
if (ret != 0) {
    // error exporting key
}
```

C.15.2.11 function wc_curve25519_export_private_raw_ex

```
int wc_curve25519_export_private_raw_ex(
    curve25519_key * key,
    byte * out,
    word32 * outLen,
```

```
    int endian
)
```

This function exports a private key from a curve25519_key structure and stores it in the given out buffer. It also sets outLen to be the size of the exported key. Can specify whether it's big or little endian.

Parameters:

- **key** Pointer to the structure from which to export the key.
- **out** Pointer to the buffer in which to store the exported key.
- **outLen** On in, is the size of the out in bytes. On out, will store the bytes written to the output buffer.
- **endian** EC25519_BIG_ENDIAN or EC25519_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve25519_init
- wc_curve25519_make_key
- wc_curve25519_import_private_raw
- wc_curve25519_export_private_raw
- wc_curve25519_size

Return:

- 0 Returned on successfully exporting the private key from the curve25519_key structure.
- BAD_FUNC_ARG Returned if any input parameters are NULL.
- ECC_BAD_ARG_E Returned if *outLen is less than wc_curve25519_size().

Example

```
int ret;

byte priv[32];
word32 privSz;
curve25519_key key;
// initialize and make key
ret = wc_curve25519_export_private_raw_ex(&key, priv, &privSz,
        EC25519_BIG_ENDIAN);
if (ret != 0) {
    // error exporting key
}
```

C.15.2.12 function wc_curve25519_import_public

```
int wc_curve25519_import_public(
    const byte * in,
    word32 inLen,
    curve25519_key * key
)
```

This function imports a public key from the given in buffer and stores it in the curve25519_key structure.

Parameters:

- **in** Pointer to the buffer containing the public key to import.
- **inLen** Length of the public key to import.
- **key** Pointer to the curve25519_key structure in which to store the key.

See:

- wc_curve25519_init
- wc_curve25519_export_public
- wc_curve25519_import_private_raw
- wc_curve25519_import_public_ex
- wc_curve25519_check_public
- wc_curve25519_size

Return:

- 0 Returned on successfully importing the public key into the curve25519_key structure.
- ECC_BAD_ARG_E Returned if the inLen parameter does not match the key size of the key structure.
- BAD_FUNC_ARG Returned if any of the input parameters are NULL.

Example

```
int ret;

byte pub[32];
// initialize pub with public key

curve25519_key key;
// initialize key

ret = wc_curve25519_import_public(pub,sizeof(pub), &key);
if (ret != 0) {
    // error importing key
}
```

C.15.2.13 function wc_curve25519_import_public_ex

```
int wc_curve25519_import_public_ex(
    const byte * in,
    word32 inLen,
    curve25519_key * key,
    int endian
)
```

This function imports a public key from the given in buffer and stores it in the curve25519_key structure.

Parameters:

- **in** Pointer to the buffer containing the public key to import.
- **inLen** Length of the public key to import.
- **key** Pointer to the curve25519_key structure in which to store the key.
- **endian** EC25519_BIG_ENDIAN or EC25519_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve25519_init
- wc_curve25519_export_public
- wc_curve25519_import_private_raw
- wc_curve25519_import_public
- wc_curve25519_check_public
- wc_curve25519_size

Return:

- 0 Returned on successfully importing the public key into the curve25519_key structure.

- ECC_BAD_ARG_E Returned if the inLen parameter does not match the key size of the key structure.
- BAD_FUNC_ARG Returned if any of the input parameters are NULL.

Example

```
int ret;

byte pub[32];
// initialize pub with public key
curve25519_key key;
// initialize key

ret = wc_curve25519_import_public_ex(pub, sizeof(pub), &key,
        EC25519_BIG_ENDIAN);
if (ret != 0) {
    // error importing key
}
```

C.15.2.14 function wc_curve25519_check_public

```
int wc_curve25519_check_public(
    const byte * pub,
    word32 pubSz,
    int endian
)
```

This function checks that a public key buffer holds a valid Curve25519 key value given the endian ordering.

Parameters:

- **pub** Pointer to the buffer containing the public key to check.
- **pubSz** Length of the public key to check.
- **endian** EC25519_BIG_ENDIAN or EC25519_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve25519_init
- wc_curve25519_import_public
- wc_curve25519_import_public_ex
- wc_curve25519_size

Return:

- 0 Returned when the public key value is valid.
- ECC_BAD_ARG_E Returned if the public key value is not valid.
- BAD_FUNC_ARG Returned if any of the input parameters are NULL.

Example

```
int ret;

byte pub[] = { Contents of public key };

ret = wc_curve25519_check_public_ex(pub, sizeof(pub), EC25519_BIG_ENDIAN);
if (ret != 0) {
    // error importing key
}
```

C.15.2.15 function wc_curve25519_export_public

```
int wc_curve25519_export_public(
    curve25519_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports a public key from the given key structure and stores the result in the out buffer. Big endian only.

Parameters:

- **key** Pointer to the curve25519_key structure in from which to export the key.
- **out** Pointer to the buffer in which to store the public key.
- **outLen** On in, is the size of the out in bytes. On out, will store the bytes written to the output buffer.

See:

- wc_curve25519_init
- wc_curve25519_export_private_raw
- wc_curve25519_import_public

Return:

- 0 Returned on successfully exporting the public key from the curve25519_key structure.
- ECC_BAD_ARG_E Returned if outLen is less than CURVE25519_PUB_KEY_SIZE.
- BAD_FUNC_ARG Returned if any of the input parameters are NULL.

Example

```
int ret;

byte pub[32];
int pubSz;

curve25519_key key;
// initialize and make key
ret = wc_curve25519_export_public(&key, pub, &pubSz);
if (ret != 0) {
    // error exporting key
}
```

C.15.2.16 function wc_curve25519_export_public_ex

```
int wc_curve25519_export_public_ex(
    curve25519_key * key,
    byte * out,
    word32 * outLen,
    int endian
)
```

This function exports a public key from the given key structure and stores the result in the out buffer. Supports both big and little endian.

Parameters:

- **key** Pointer to the curve25519_key structure in from which to export the key.
- **out** Pointer to the buffer in which to store the public key.

- **outLen** On in, is the size of the out in bytes. On out, will store the bytes written to the output buffer.
- **endian** EC25519_BIG_ENDIAN or EC25519_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve25519_init
- wc_curve25519_export_private_raw
- wc_curve25519_import_public

Return:

- 0 Returned on successfully exporting the public key from the curve25519_key structure.
- ECC_BAD_ARG_E Returned if outLen is less than CURVE25519_PUB_KEY_SIZE.
- BAD_FUNC_ARG Returned if any of the input parameters are NULL.
- NO_VALID_DEVID Returned when the public point still has to be derived, the build is WOLF_CRYPTO_CB_ONLY_CURVE25519 and the key is not bound to a device. Such a key has no software path to fall back on and is never offloaded to an arbitrary registered device, so it must be created with wc_curve25519_init_ex() and a real devId.

Example

```
int ret;

byte pub[32];
int pubSz;

curve25519_key key;
// initialize and make key

ret = wc_curve25519_export_public_ex(&key, pub, &pubSz, EC25519_BIG_ENDIAN);
if (ret != 0) {
    // error exporting key
}
```

C.15.2.17 function wc_curve25519_export_key_raw

```
int wc_curve25519_export_key_raw(
    curve25519_key * key,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz
)
```

Export Curve25519 key pair. Big endian only.

Parameters:

- **key** Pointer to the curve25519_key structure in from which to export the key pair.
- **priv** Pointer to the buffer in which to store the private key.
- **privSz** On in, is the size of the priv buffer in bytes. On out, will store the bytes written to the priv buffer.
- **pub** Pointer to the buffer in which to store the public key.
- **pubSz** On in, is the size of the pub buffer in bytes. On out, will store the bytes written to the pub buffer.

See:

- wc_curve25519_export_key_raw_ex

- wc_curve25519_export_private_raw

Return:

- 0 Returned on successfully exporting the key pair from the curve25519_key structure.
- BAD_FUNC_ARG Returned if any input parameters are NULL.
- ECC_BAD_ARG_E Returned if privSz is less than CURVE25519_KEY_SIZE or pubSz is less than CURVE25519_PUB_KEY_SIZE.

Example

```
int ret;

byte pub[32];
byte priv[32];
int pubSz;
int privSz;

curve25519_key key;
// initialize and make key

ret = wc_curve25519_export_key_raw(&key, priv, &privSz, pub, &pubSz);
if (ret != 0) {
    // error exporting key
}
```

C.15.2.18 function wc_curve25519_export_key_raw_ex

```
int wc_curve25519_export_key_raw_ex(
    curve25519_key * key,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz,
    int endian
)
```

Export curve25519 key pair. Big or little endian.

Parameters:

- **key** Pointer to the curve25519_key structure in from which to export the key pair.
- **priv** Pointer to the buffer in which to store the private key.
- **privSz** On in, is the size of the priv buffer in bytes. On out, will store the bytes written to the priv buffer.
- **pub** Pointer to the buffer in which to store the public key.
- **pubSz** On in, is the size of the pub buffer in bytes. On out, will store the bytes written to the pub buffer.
- **endian** EC25519_BIG_ENDIAN or EC25519_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve25519_export_key_raw
- wc_curve25519_export_private_raw_ex
- wc_curve25519_export_public_ex

Return:

- 0 Returned on successfully exporting the key pair from the curve25519_key structure.
- BAD_FUNC_ARG Returned if any input parameters are NULL.

- ECC_BAD_ARG_E Returned if privSz is less than CURVE25519_KEY_SIZE or pubSz is less than CURVE25519_PUB_KEY_SIZE.

Example

```
int ret;

byte pub[32];
byte priv[32];
int pubSz;
int privSz;

curve25519_key key;
// initialize and make key

ret = wc_curve25519_export_key_raw_ex(&key, priv, &privSz, pub, &pubSz,
        EC25519_BIG_ENDIAN);
if (ret != 0) {
    // error exporting key
}
```

C.15.2.19 function wc_curve25519_size

```
int wc_curve25519_size(
    curve25519_key * key
)
```

This function returns the key size of the given key structure.

Parameters:

- **key** Pointer to the curve25519_key structure in for which to determine the key size.

See:

- wc_curve25519_init
- wc_curve25519_make_key

Return:

- Success Given a valid, initialized curve25519_key structure, returns the size of the key.
- 0 Returned if key is NULL

Example

```
int keySz;

curve25519_key key;
// initialize and make key

keySz = wc_curve25519_size(&key);
```

C.15.2.20 function wc_curve25519_make_pub

```
int wc_curve25519_make_pub(
    int public_size,
    byte * pub,
    int private_size,
    const byte * priv
)
```

This function generates a Curve25519 public key from a given private key. This is a lower-level function that operates directly on byte buffers rather than curve25519_key structures.

Parameters:

- **public_size** Size of the public key buffer (must be 32)
- **pub** Pointer to buffer to store the public key
- **private_size** Size of the private key (must be 32)
- **priv** Pointer to buffer containing the private key

See:

- wc_curve25519_make_key
- wc_curve25519_make_pub_blind

Return:

- 0 On successfully generating the public key
- ECC_BAD_ARG_E If the key sizes are invalid
- BAD_FUNC_ARG If any input parameters are NULL

Example

```
byte priv[CURVE25519_KEYSIZE];
byte pub[CURVE25519_KEYSIZE];

// initialize priv with private key
int ret = wc_curve25519_make_pub(sizeof(pub), pub, sizeof(priv),
                                 priv);
if (ret != 0) {
    // error generating public key
}
```

C.15.2.21 function wc_curve25519_make_pub_blind

```
int wc_curve25519_make_pub_blind(
    int public_size,
    byte * pub,
    int private_size,
    const byte * priv,
    WC_RNG * rng
)
```

This function generates a Curve25519 public key from a given private key with blinding to resist side-channel attacks. This adds randomization to the scalar multiplication operation.

Parameters:

- **public_size** Size of the public key buffer (must be 32)
- **pub** Pointer to buffer to store the public key
- **private_size** Size of the private key (must be 32)
- **priv** Pointer to buffer containing the private key
- **rng** Pointer to initialized RNG for blinding

See:

- wc_curve25519_make_pub
- wc_curve25519_generic_blind

Return:

- 0 On successfully generating the public key

- ECC_BAD_ARG_E If the key sizes are invalid
- BAD_FUNC_ARG If any input parameters are NULL

Example

```
WC_RNG rng;
byte priv[CURVE25519_KEYSIZE];
byte pub[CURVE25519_KEYSIZE];

wc_InitRng(&rng);
// initialize priv with private key
int ret = wc_curve25519_make_pub_blind(sizeof(pub), pub,
                                       sizeof(priv), priv, &rng);
if (ret != 0) {
    // error generating public key
}
```

C.15.2.22 function wc_curve25519_generic

```
int wc_curve25519_generic(
    int public_size,
    byte * pub,
    int private_size,
    const byte * priv,
    int basepoint_size,
    const byte * basepoint
)
```

This function performs a generic Curve25519 scalar multiplication with a custom basepoint. This allows computing scalar * basepoint for any basepoint, not just the standard generator.

Parameters:

- **public_size** Size of the output buffer (must be 32)
- **pub** Pointer to buffer to store the result
- **private_size** Size of the scalar (must be 32)
- **priv** Pointer to buffer containing the scalar
- **basepoint_size** Size of the basepoint (must be 32)
- **basepoint** Pointer to buffer containing the basepoint

See:

- wc_curve25519_shared_secret
- wc_curve25519_generic_blind

Return:

- 0 On successfully computing the result
- ECC_BAD_ARG_E If the sizes are invalid
- BAD_FUNC_ARG If any input parameters are NULL

Example

```
byte scalar[CURVE25519_KEYSIZE];
byte basepoint[CURVE25519_KEYSIZE];
byte result[CURVE25519_KEYSIZE];

// initialize scalar and basepoint
int ret = wc_curve25519_generic(sizeof(result), result,
                                sizeof(scalar), scalar,
```

```
                                    sizeof(basepoint), basepoint);
if (ret != 0) {
    // error computing result
}
```

C.15.2.23 function wc_curve25519_generic_blind

```
int wc_curve25519_generic_blind(
    int public_size,
    byte * pub,
    int private_size,
    const byte * priv,
    int basepoint_size,
    const byte * basepoint,
    WC_RNG * rng
)
```

This function performs a generic Curve25519 scalar multiplication with a custom basepoint and blinding to resist side-channel attacks.

Parameters:

- **public_size** Size of the output buffer (must be 32)
- **pub** Pointer to buffer to store the result
- **private_size** Size of the scalar (must be 32)
- **priv** Pointer to buffer containing the scalar
- **basepoint_size** Size of the basepoint (must be 32)
- **basepoint** Pointer to buffer containing the basepoint
- **rng** Pointer to initialized RNG for blinding

See:

- wc_curve25519_generic
- wc_curve25519_make_pub_blind

Return:

- 0 On successfully computing the result
- ECC_BAD_ARG_E If the sizes are invalid
- BAD_FUNC_ARG If any input parameters are NULL

Example

```
WC_RNG rng;
byte scalar[CURVE25519_KEYSIZE];
byte basepoint[CURVE25519_KEYSIZE];
byte result[CURVE25519_KEYSIZE];

wc_InitRng(&rng);
// initialize scalar and basepoint
int ret = wc_curve25519_generic_blind(sizeof(result), result,
                                      sizeof(scalar), scalar,
                                      sizeof(basepoint), basepoint,
                                      &rng);
```

C.15.2.24 function wc_curve25519_make_priv

```
int wc_curve25519_make_priv(
    WC_RNG * rng,
```

```
    int keysize,
    byte * priv
)
```

This function generates a Curve25519 private key using the given random number generator. This is a lower-level function that generates only the private key bytes.

Parameters:

- **rng** Pointer to initialized RNG
- **keysize** Size of the key to generate (must be 32)
- **priv** Pointer to buffer to store the private key

See:

- wc_curve25519_make_key
- wc_curve25519_make_pub

Return:

- 0 On successfully generating the private key
- ECC_BAD_ARG_E If keysize is invalid
- BAD_FUNC_ARG If any input parameters are NULL
- RNG_FAILURE_E If random number generation fails

Example

```
WC_RNG rng;
byte priv[CURVE25519_KEYSIZE];

wc_InitRng(&rng);
int ret = wc_curve25519_make_priv(&rng, sizeof(priv), priv);
if (ret != 0) {
    // error generating private key
}
```

C.15.2.25 function wc_curve25519_init_ex

```
int wc_curve25519_init_ex(
    curve25519_key * key,
    void * heap,
    int devId
)
```

This function initializes a Curve25519 key with extended parameters, allowing specification of custom heap and device ID for hardware acceleration.

Parameters:

- **key** Pointer to the curve25519_key structure to initialize
- **heap** Pointer to heap hint for memory allocation (can be NULL)
- **devId** Device ID for hardware acceleration (use INVALID_DEVID for software only)

See:

- wc_curve25519_init
- wc_curve25519_free

Return:

- 0 On successfully initializing the key
- BAD_FUNC_ARG If key is NULL

Example

```
curve25519_key key;
void* heap = NULL;
int devId = INVALID_DEVID;

int ret = wc_curve25519_init_ex(&key, heap, devId);
if (ret != 0) {
    // error initializing key
}
```

C.15.2.26 function wc_curve25519_set_rng

```
int wc_curve25519_set_rng(
    curve25519_key * key,
    WC_RNG * rng
)
```

This function sets the RNG to be used with a Curve25519 key. This is useful for operations that require randomness such as blinded scalar multiplication.

Parameters:

- **key** Pointer to the curve25519_key structure
- **rng** Pointer to initialized RNG

See:

- wc_curve25519_init
- wc_curve25519_make_key

Return:

- 0 On successfully setting the RNG
- BAD_FUNC_ARG If key or rng is NULL

Example

```
WC_RNG rng;
curve25519_key key;

wc_InitRng(&rng);
wc_curve25519_init(&key);
int ret = wc_curve25519_set_rng(&key, &rng);
if (ret != 0) {
    // error setting RNG
}
```

C.15.2.27 function wc_curve25519_new

```
curve25519_key * wc_curve25519_new(
    void * heap,
    int devId,
    int * result_code
)
```

This function allocates and initializes a new Curve25519 key structure with extended parameters. The caller is responsible for freeing the key with wc_curve25519_delete. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.

Parameters:

- **heap** Pointer to heap hint for memory allocation (can be NULL)
- **devId** Device ID for hardware acceleration (use INVALID_DEVID for software only)
- **result_code** Pointer to store result code (0 on success)

See:

- wc_curve25519_delete
- wc_curve25519_init_ex

Return:

- Pointer to newly allocated curve25519_key on success
- NULL on failure

Note: This API is only available when WC_NO_CONSTRUCTORS is not defined. WC_NO_CONSTRUCTORS is automatically defined when WOLFSSL_NO_MALLOC is defined.

Example

```
int ret;
curve25519_key* key;

key = wc_curve25519_new(NULL, INVALID_DEVID, &ret);
if (key == NULL || ret != 0) {
    // error allocating key
}
// use key
wc_curve25519_delete(key, &key);
```

C.15.2.28 function wc_curve25519_delete

```
int wc_curve25519_delete(
    curve25519_key * key,
    curve25519_key ** key_p
)
```

This function frees a Curve25519 key structure that was allocated with wc_curve25519_new and sets the pointer to NULL. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.

Parameters:

- **key** Pointer to the curve25519_key structure to free
- **key_p** Pointer to the key pointer (will be set to NULL)

See:

- wc_curve25519_new
- wc_curve25519_free

Return:

- 0 On successfully freeing the key
- BAD_FUNC_ARG If key or key_p is NULL

Note: This API is only available when WC_NO_CONSTRUCTORS is not defined. WC_NO_CONSTRUCTORS is automatically defined when WOLFSSL_NO_MALLOC is defined.

Example

```
int ret;
curve25519_key* key;

key = wc_curve25519_new(NULL, INVALID_DEVID, &ret);
// use key
ret = wc_curve25519_delete(key, &key);
if (ret != 0) {
    // error freeing key
}
// key is now NULL
```

C.15.3 Source code

```
int wc_curve25519_make_key(WC_RNG* rng, int keysize, curve25519_key* key);

int wc_curve25519_shared_secret(curve25519_key* private_key,
                                curve25519_key* public_key,
                                byte* out, word32* outlen);

int wc_curve25519_shared_secret_ex(curve25519_key* private_key,
                                   curve25519_key* public_key,
                                   byte* out, word32* outlen, int endian);

int wc_curve25519_init(curve25519_key* key);

void wc_curve25519_free(curve25519_key* key);

int wc_curve25519_import_private(const byte* priv, word32 privSz,
                                 curve25519_key* key);

int wc_curve25519_import_private_ex(const byte* priv, word32 privSz,
                                    curve25519_key* key, int endian);

int wc_curve25519_import_private_raw(const byte* priv, word32 privSz,
                            const byte* pub, word32 pubSz, curve25519_key* key);

int wc_curve25519_import_private_raw_ex(const byte* priv, word32 privSz,
                                       const byte* pub, word32 pubSz,
                                       curve25519_key* key, int endian);

int wc_curve25519_export_private_raw(curve25519_key* key, byte* out,
                                     word32* outLen);

int wc_curve25519_export_private_raw_ex(curve25519_key* key, byte* out,
                                       word32* outLen, int endian);

int wc_curve25519_import_public(const byte* in, word32 inLen,
                                curve25519_key* key);

int wc_curve25519_import_public_ex(const byte* in, word32 inLen,
                                   curve25519_key* key, int endian);

int wc_curve25519_check_public(const byte* pub, word32 pubSz, int endian);
```

```
int wc_curve25519_export_public(curve25519_key* key, byte* out, word32*
 ↪ outLen);

int wc_curve25519_export_public_ex(curve25519_key* key, byte* out,
                                   word32* outLen, int endian);

int wc_curve25519_export_key_raw(curve25519_key* key,
                                 byte* priv, word32 *privSz,
                                 byte* pub, word32 *pubSz);

int wc_curve25519_export_key_raw_ex(curve25519_key* key,
                                    byte* priv, word32 *privSz,
                                    byte* pub, word32 *pubSz,
                                    int endian);

int wc_curve25519_size(curve25519_key* key);

int wc_curve25519_make_pub(int public_size, byte* pub, int private_size,
                           const byte* priv);

int wc_curve25519_make_pub_blind(int public_size, byte* pub,
                                 int private_size, const byte* priv,
                                 WC_RNG* rng);

int wc_curve25519_generic(int public_size, byte* pub, int private_size,
                          const byte* priv, int basepoint_size,
                          const byte* basepoint);

int wc_curve25519_generic_blind(int public_size, byte* pub,
                                int private_size, const byte* priv,
                                int basepoint_size, const byte* basepoint,
                                WC_RNG* rng);

int wc_curve25519_make_priv(WC_RNG* rng, int keysize, byte* priv);

int wc_curve25519_init_ex(curve25519_key* key, void* heap, int devId);

int wc_curve25519_set_rng(curve25519_key* key, WC_RNG* rng);

curve25519_key* wc_curve25519_new(void* heap, int devId,
                                  int *result_code);

int wc_curve25519_delete(curve25519_key* key, curve25519_key** key_p);
```

C.16 curve448.h

C.16.1 Functions

	Name
int	**wc_curve448_make_key**(WC_RNG * rng, int keysize, curve448_key * key)This function generates a Curve448 key using the given random number generator, rng, of the size given (keysize), and stores it in the given curve448_key structure. It should be called after the key structure has been initialized through wc_curve448_init().
int	**wc_curve448_shared_secret**(curve448_key * private_key, curve448_key * public_key, byte * out, word32 * outlen)This function computes a shared secret key given a secret private key and a received public key. It stores the generated secret key in the buffer out and assigns the variable of the secret key to outlen. Only supports big endian.
int	**wc_curve448_shared_secret_ex**(curve448_key * private_key, curve448_key * public_key, byte * out, word32 * outlen, int endian)This function computes a shared secret key given a secret private key and a received public key. It stores the generated secret key in the buffer out and assigns the variable of the secret key to outlen. Supports both big and little endian.
int	**wc_curve448_init**(curve448_key * key)This function initializes a Curve448 key. It should be called before generating a key for the structure.
int	**wc_curve448_init_ex**(curve448_key * key, void * heap, int devId)This function initializes a Curve448 key with extended parameters, allowing specification of custom heap and device ID for hardware acceleration.
curve448_key *	**wc_curve448_new**(void * heap, int devId, int * result_code)This function allocates and initializes a new Curve448 key structure with extended parameters. The caller is responsible for freeing the key with wc_curve448_delete. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.
int	**wc_curve448_delete**(curve448_key * key, curve448_key ** key_p)This function frees a Curve448 key structure that was allocated with wc_curve448_new and sets the pointer to NULL. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.
void	**wc_curve448_free**(curve448_key * key)This function frees a Curve448 object.

	Name
int	**wc_curve448_import_private**(const byte * priv, word32 privSz, curve448_key * key)This function imports a curve448 private key only. (Big endian).
int	**wc_curve448_import_private_ex**(const byte * priv, word32 privSz, curve448_key * key, int endian)curve448 private key import only. (Big or Little endian).
int	**wc_curve448_import_private_raw**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, curve448_key * key)This function imports a public-private key pair into a curve448_key structure. Big endian only.
int	**wc_curve448_import_private_raw_ex**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, curve448_key * key, int endian)This function imports a public-private key pair into a curve448_key structure. Supports both big and little endian.
int	**wc_curve448_export_private_raw**(curve448_key * key, byte * out, word32 * outLen)This function exports a private key from a curve448_key structure and stores it in the given out buffer. It also sets outLen to be the size of the exported key. Big Endian only.
int	**wc_curve448_export_private_raw_ex**(curve448_key * key, byte * out, word32 * outLen, int endian)This function exports a private key from a curve448_key structure and stores it in the given out buffer. It also sets outLen to be the size of the exported key. Can specify whether it's big or little endian.
int	**wc_curve448_import_public**(const byte * in, word32 inLen, curve448_key * key)This function imports a public key from the given in buffer and stores it in the curve448_key structure.
int	**wc_curve448_import_public_ex**(const byte * in, word32 inLen, curve448_key * key, int endian)This function imports a public key from the given in buffer and stores it in the curve448_key structure.
int	**wc_curve448_check_public**(const byte * pub, word32 pubSz, int endian)This function checks that a public key buffer holds a valid Curve448 key value given the endian ordering.
int	**wc_curve448_export_public**(curve448_key * key, byte * out, word32 * outLen)This function exports a public key from the given key structure and stores the result in the out buffer. Big endian only.

	Name
int	**wc_curve448_export_public_ex**(curve448_key * key, byte * out, word32 * outLen, int endian)This function exports a public key from the given key structure and stores the result in the out buffer. Supports both big and little endian.
int	**wc_curve448_export_key_raw**(curve448_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz)This function exports a key pair from the given key structure and stores the result in the out buffer. Big endian only.
int	**wc_curve448_export_key_raw_ex**(curve448_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz, int endian)Export curve448 key pair. Big or little endian.
int	**wc_curve448_size**(curve448_key * key)This function returns the key size of the given key structure.
int	**wc_curve448_make_pub**(int public_size, byte * pub, int private_size, const byte * priv)This function generates a Curve448 public key from a given private key. It computes the public key by performing scalar multiplication of the base point with the private key.
int	**wc_curve448_generic**(int public_size, byte * pub, int private_size, const byte * priv, int basepoint_size, const byte * basepoint)This function performs a generic Curve448 scalar multiplication with a custom basepoint. This allows computing scalar * basepoint for any basepoint, not just the standard generator. As in wc_curve448_shared_secret_ex, an all_zero result is rejected (RFC 7748 section 6.2); build with WOLFSSL_NO_ECDHX_SHARED_ZERO_CHECK to get the raw scalar multiplication result instead.

C.16.2 Functions Documentation

C.16.2.1 function wc_curve448_make_key

```
int wc_curve448_make_key(
    WC_RNG * rng,
    int keysize,
    curve448_key * key
)
```

This function generates a Curve448 key using the given random number generator, rng, of the size given (keysize), and stores it in the given curve448_key structure. It should be called after the key structure has been initialized through wc_curve448_init().

Parameters:

- **rng** Pointer to the RNG object used to generate the ecc key.

- **keysize** Size of the key to generate. Must be 56 bytes for curve448.
- **key** Pointer to the curve448_key structure in which to store the generated key.

See: wc_curve448_init

Return:

- 0 Returned on successfully generating the key and and storing it in the given curve448_key structure.
- ECC_BAD_ARG_E Returned if the input keysize does not correspond to the keysize for a curve448 key (56 bytes).
- RNG_FAILURE_E Returned if the rng internal status is not DRBG_OK or if there is in generating the next random block with rng.
- BAD_FUNC_ARG Returned if any of the input parameters passed in are NULL.

Example

```
int ret;

curve448_key key;
wc_curve448_init(&key); // initialize key
WC_RNG rng;
wc_InitRng(&rng); // initialize random number generator

ret = wc_curve448_make_key(&rng, 56, &key);
if (ret != 0) {
    // error making Curve448 key
}
```

C.16.2.2 function wc_curve448_shared_secret

```
int wc_curve448_shared_secret(
    curve448_key * private_key,
    curve448_key * public_key,
    byte * out,
    word32 * outlen
)
```

This function computes a shared secret key given a secret private key and a received public key. It stores the generated secret key in the buffer out and assigns the variable of the secret key to outlen. Only supports big endian.

Parameters:

- **private_key** Pointer to the curve448_key structure initialized with the user's private key.
- **public_key** Pointer to the curve448_key structure containing the received public key.
- **out** Pointer to a buffer in which to store the 56 byte computed secret key.
- **outlen** Pointer in which to store the length written to the output buffer.

See:

- wc_curve448_init
- wc_curve448_make_key
- wc_curve448_shared_secret_ex

Return:

- 0 Returned on successfully computing a shared secret key
- BAD_FUNC_ARG Returned if any of the input parameters passed in are NULL

Example

```
int ret;

byte sharedKey[56];
word32 keySz;
curve448_key privKey, pubKey;
// initialize both keys

ret = wc_curve448_shared_secret(&privKey, &pubKey, sharedKey, &keySz);
if (ret != 0) {
    // error generating shared key
}
```

C.16.2.3 function wc_curve448_shared_secret_ex

```
int wc_curve448_shared_secret_ex(
    curve448_key * private_key,
    curve448_key * public_key,
    byte * out,
    word32 * outlen,
    int endian
)
```

This function computes a shared secret key given a secret private key and a received public key. It stores the generated secret key in the buffer out and assigns the variable of the secret key to outlen. Supports both big and little endian.

Parameters:

- **private_key** Pointer to the curve448_key structure initialized with the user's private key.
- **public_key** Pointer to the curve448_key structure containing the received public key.
- **out** Pointer to a buffer in which to store the 56 byte computed secret key.
- **outlen** Pointer in which to store the length written to the output buffer.
- **endian** EC448_BIG_ENDIAN or EC448_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve448_init
- wc_curve448_make_key
- wc_curve448_shared_secret

Return:

- 0 Returned on successfully computing a shared secret key.
- BAD_FUNC_ARG Returned if any of the input parameters passed in are NULL.

Example

```
int ret;

byte sharedKey[56];
word32 keySz;

curve448_key privKey, pubKey;
// initialize both keys

ret = wc_curve448_shared_secret_ex(&privKey, &pubKey, sharedKey, &keySz,
        EC448_BIG_ENDIAN);
if (ret != 0) {
```

```
    // error generating shared key
}
```

C.16.2.4 function wc_curve448_init

```
int wc_curve448_init(
    curve448_key * key
)
```

This function initializes a Curve448 key. It should be called before generating a key for the structure.

Parameters:

- **key** Pointer to the curve448_key structure to initialize.

See: wc_curve448_make_key

Return:

- 0 Returned on successfully initializing the curve448_key structure.
- BAD_FUNC_ARG Returned when key is NULL.

Example

```
curve448_key key;
wc_curve448_init(&key); // initialize key
// make key and proceed to encryption
```

C.16.2.5 function wc_curve448_init_ex

```
int wc_curve448_init_ex(
    curve448_key * key,
    void * heap,
    int devId
)
```

This function initializes a Curve448 key with extended parameters, allowing specification of custom heap and device ID for hardware acceleration.

Parameters:

- **key** Pointer to the curve448_key structure to initialize.
- **heap** Pointer to heap hint for memory allocation (can be NULL)
- **devId** Device ID for hardware acceleration (use INVALID_DEVID for software only)

See:

- wc_curve448_init
- wc_curve448_free

Return:

- 0 On successfully initializing the key
- BAD_FUNC_ARG If key is NULL

Example

```
curve448_key key;
void* heap = NULL;
int devId = INVALID_DEVID;

int ret = wc_curve448_init_ex(&key, heap, devId);
if (ret != 0) {
```

```
    // error initializing key
}
```

C.16.2.6 function wc_curve448_new

```
curve448_key * wc_curve448_new(
    void * heap,
    int devId,
    int * result_code
)
```

This function allocates and initializes a new Curve448 key structure with extended parameters. The caller is responsible for freeing the key with wc_curve448_delete. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.

Parameters:

- **heap** Pointer to heap hint for memory allocation (can be NULL)
- **devId** Device ID for hardware acceleration (use INVALID_DEVID for software only)
- **result_code** Pointer to store result code (0 on success)

See:

- wc_curve448_delete
- wc_curve448_init_ex

Return:

- Pointer to newly allocated curve448_key on success
- NULL on failure

Note: This API is only available when WC_NO_CONSTRUCTORS is not defined. WC_NO_CONSTRUCTORS is automatically defined when WOLFSSL_NO_MALLOC is defined.

Example

```
int ret;
curve448_key* key;

key = wc_curve448_new(NULL, INVALID_DEVID, &ret);
if (key == NULL || ret != 0) {
    // error allocating key
}
// use key
wc_curve448_delete(key, &key);
```

C.16.2.7 function wc_curve448_delete

```
int wc_curve448_delete(
    curve448_key * key,
    curve448_key ** key_p
)
```

This function frees a Curve448 key structure that was allocated with wc_curve448_new and sets the pointer to NULL. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.

Parameters:

- **key** Pointer to the curve448_key structure to free.

- **key_p** Pointer to the key pointer to set to NULL (can be NULL)

See:

- wc_curve448_new
- wc_curve448_free

Return:

- 0 On successfully freeing the key
- BAD_FUNC_ARG If key is NULL

Note: This API is only available when WC_NO_CONSTRUCTORS is not defined. WC_NO_CONSTRUCTORS is automatically defined when WOLFSSL_NO_MALLOC is defined.

Example

```
int ret;
curve448_key* key = wc_curve448_new(NULL, INVALID_DEVID, &ret);
// use key
ret = wc_curve448_delete(key, &key);
```

C.16.2.8 function wc_curve448_free

```
void wc_curve448_free(
    curve448_key * key
)
```

This function frees a Curve448 object.

Parameters:

- **key** Pointer to the key object to free.

See:

- wc_curve448_init
- wc_curve448_make_key

Example

```
curve448_key privKey;
// initialize key, use it to generate shared secret key
wc_curve448_free(&privKey);
```

C.16.2.9 function wc_curve448_import_private

```
int wc_curve448_import_private(
    const byte * priv,
    word32 privSz,
    curve448_key * key
)
```

This function imports a curve448 private key only. (Big endian).

Parameters:

- **priv** Pointer to a buffer containing the private key to import.
- **privSz** Length of the private key to import.
- **key** Pointer to the structure in which to store the imported key.

See:

- wc_curve448_import_private_ex

- wc_curve448_size

Return:

- 0 Returned on successfully importing private key.
- BAD_FUNC_ARG Returns if key or priv is null.
- ECC_BAD_ARG_E Returns if privSz is not equal to CURVE448_KEY_SIZE.

Example

```
int ret;

byte priv[] = { Contents of private key };
curve448_key key;
wc_curve448_init(&key);

ret = wc_curve448_import_private(priv, sizeof(priv), &key);
if (ret != 0) {
    // error importing key
}
```

C.16.2.10 function wc_curve448_import_private_ex

```
int wc_curve448_import_private_ex(
    const byte * priv,
    word32 privSz,
    curve448_key * key,
    int endian
)
```

curve448 private key import only. (Big or Little endian).

Parameters:

- **priv** Pointer to a buffer containing the private key to import.
- **privSz** Length of the private key to import.
- **key** Pointer to the structure in which to store the imported key.
- **endian** EC448_BIG_ENDIAN or EC448_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve448_import_private
- wc_curve448_size

Return:

- 0 Returned on successfully importing private key.
- BAD_FUNC_ARG Returns if key or priv is null.
- ECC_BAD_ARG_E Returns if privSz is not equal to CURVE448_KEY_SIZE.

Example

```
int ret;

byte priv[] = { // Contents of private key };
curve448_key key;
wc_curve448_init(&key);

ret = wc_curve448_import_private_ex(priv, sizeof(priv), &key,
        EC448_BIG_ENDIAN);
if (ret != 0) {
```

```
    // error importing key
}
```

C.16.2.11 function wc_curve448_import_private_raw

```
int wc_curve448_import_private_raw(
    const byte * priv,
    word32 privSz,
    const byte * pub,
    word32 pubSz,
    curve448_key * key
)
```

This function imports a public-private key pair into a curve448_key structure. Big endian only.

Parameters:

- **priv** Pointer to a buffer containing the private key to import.
- **privSz** Length of the private key to import.
- **pub** Pointer to a buffer containing the public key to import.
- **pubSz** Length of the public key to import.
- **key** Pointer to the structure in which to store the imported keys

See:

- wc_curve448_init
- wc_curve448_make_key
- wc_curve448_import_public
- wc_curve448_export_private_raw

Return:

- 0 Returned on importing into the curve448_key structure.
- BAD_FUNC_ARG Returns if any of the input parameters are null.
- ECC_BAD_ARG_E Returned if the input key's key size does not match the public or private key sizes.

Example

```
int ret;

byte priv[56];
byte pub[56];
// initialize with public and private keys
curve448_key key;

wc_curve448_init(&key);
// initialize key

ret = wc_curve448_import_private_raw(&priv, sizeof(priv), pub, sizeof(pub),
        &key);
if (ret != 0) {
    // error importing keys
}
```

C.16.2.12 function wc_curve448_import_private_raw_ex

```
int wc_curve448_import_private_raw_ex(
    const byte * priv,
```

```
    word32 privSz,
    const byte * pub,
    word32 pubSz,
    curve448_key * key,
    int endian
)
```

This function imports a public-private key pair into a curve448_key structure. Supports both big and little endian.

Parameters:

- **priv** Pointer to a buffer containing the private key to import.
- **privSz** Length of the private key to import.
- **pub** Pointer to a buffer containing the public key to import.
- **pubSz** Length of the public key to import.
- **key** Pointer to the structure in which to store the imported keys.
- **endian** EC448_BIG_ENDIAN or EC448_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve448_init
- wc_curve448_make_key
- wc_curve448_import_public
- wc_curve448_export_private_raw
- wc_curve448_import_private_raw

Return:

- 0 Returned on importing into the curve448_key structure.
- BAD_FUNC_ARG Returns if any of the input parameters are null.
- ECC_BAD_ARG_E Returned if the input key's key size does not match the public or private key sizes.

Example

```
int ret;

byte priv[56];
byte pub[56];
// initialize with public and private keys
curve448_key key;

wc_curve448_init(&key);
// initialize key

ret = wc_curve448_import_private_raw_ex(&priv, sizeof(priv), pub,
        sizeof(pub), &key, EC448_BIG_ENDIAN);
if (ret != 0) {
    // error importing keys
}
```

C.16.2.13 function wc_curve448_export_private_raw

```
int wc_curve448_export_private_raw(
    curve448_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports a private key from a curve448_key structure and stores it in the given out buffer. It also sets outLen to be the size of the exported key. Big Endian only.

Parameters:

- **key** Pointer to the structure from which to export the key.
- **out** Pointer to the buffer in which to store the exported key.
- **outLen** On in, is the size of the out in bytes. On out, will store the bytes written to the output buffer.

See:

- wc_curve448_init
- wc_curve448_make_key
- wc_curve448_import_private_raw
- wc_curve448_export_private_raw_ex

Return:

- 0 Returned on successfully exporting the private key from the curve448_key structure.
- BAD_FUNC_ARG Returned if any input parameters are NULL.
- ECC_BAD_ARG_E Returned if wc_curve448_size() is not equal to key.

Example

```
int ret;
byte priv[56];
int privSz;

curve448_key key;
// initialize and make key

ret = wc_curve448_export_private_raw(&key, priv, &privSz);
if (ret != 0) {
    // error exporting key
}
```

C.16.2.14 function wc_curve448_export_private_raw_ex

```
int wc_curve448_export_private_raw_ex(
    curve448_key * key,
    byte * out,
    word32 * outLen,
    int endian
)
```

This function exports a private key from a curve448_key structure and stores it in the given out buffer. It also sets outLen to be the size of the exported key. Can specify whether it's big or little endian.

Parameters:

- **key** Pointer to the structure from which to export the key.
- **out** Pointer to the buffer in which to store the exported key.
- **outLen** On in, is the size of the out in bytes. On out, will store the bytes written to the output buffer.
- **endian** EC448_BIG_ENDIAN or EC448_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve448_init
- wc_curve448_make_key

- wc_curve448_import_private_raw
- wc_curve448_export_private_raw
- wc_curve448_size

Return:

- 0 Returned on successfully exporting the private key from the curve448_key structure.
- BAD_FUNC_ARG Returned if any input parameters are NULL.
- ECC_BAD_ARG_E Returned if wc_curve448_size() is not equal to key.

Example

```
int ret;

byte priv[56];
int privSz;
curve448_key key;
// initialize and make key
ret = wc_curve448_export_private_raw_ex(&key, priv, &privSz,
        EC448_BIG_ENDIAN);
if (ret != 0) {
    // error exporting key
}
```

C.16.2.15 function wc_curve448_import_public

```
int wc_curve448_import_public(
    const byte * in,
    word32 inLen,
    curve448_key * key
)
```

This function imports a public key from the given in buffer and stores it in the curve448_key structure.

Parameters:

- **in** Pointer to the buffer containing the public key to import.
- **inLen** Length of the public key to import.
- **key** Pointer to the curve448_key structure in which to store the key.

See:

- wc_curve448_init
- wc_curve448_export_public
- wc_curve448_import_private_raw
- wc_curve448_import_public_ex
- wc_curve448_check_public
- wc_curve448_size

Return:

- 0 Returned on successfully importing the public key into the curve448_key structure.
- ECC_BAD_ARG_E Returned if the inLen parameter does not match the key size of the key structure.
- BAD_FUNC_ARG Returned if any of the input parameters are NULL.

Example

```
int ret;

byte pub[56];
```

```
// initialize pub with public key

curve448_key key;
// initialize key

ret = wc_curve448_import_public(pub,sizeof(pub), &key);
if (ret != 0) {
    // error importing key
}
```

C.16.2.16 function wc_curve448_import_public_ex

```
int wc_curve448_import_public_ex(
    const byte * in,
    word32 inLen,
    curve448_key * key,
    int endian
)
```

This function imports a public key from the given in buffer and stores it in the curve448_key structure.

Parameters:

- **in** Pointer to the buffer containing the public key to import.
- **inLen** Length of the public key to import.
- **key** Pointer to the curve448_key structure in which to store the key.
- **endian** EC448_BIG_ENDIAN or EC448_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve448_init
- wc_curve448_export_public
- wc_curve448_import_private_raw
- wc_curve448_import_public
- wc_curve448_check_public
- wc_curve448_size

Return:

- 0 Returned on successfully importing the public key into the curve448_key structure.
- ECC_BAD_ARG_E Returned if the inLen parameter does not match the key size of the key structure.
- BAD_FUNC_ARG Returned if any of the input parameters are NULL.

Example

```
int ret;

byte pub[56];
// initialize pub with public key
curve448_key key;
// initialize key

ret = wc_curve448_import_public_ex(pub, sizeof(pub), &key,
        EC448_BIG_ENDIAN);
if (ret != 0) {
    // error importing key
}
```

C.16.2.17 function wc_curve448_check_public

```
int wc_curve448_check_public(
    const byte * pub,
    word32 pubSz,
    int endian
)
```

This function checks that a public key buffer holds a valid Curve448 key value given the endian ordering.

Parameters:

- **pub** Pointer to the buffer containing the public key to check.
- **pubSz** Length of the public key to check.
- **endian** EC448_BIG_ENDIAN or EC448_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve448_init
- wc_curve448_import_public
- wc_curve448_import_public_ex
- wc_curve448_size

Return:

- 0 Returned when the public key value is valid.
- ECC_BAD_ARG_E Returned if the public key value is not valid.
- BAD_FUNC_ARG Returned if any of the input parameters are NULL.

Example

```
int ret;

byte pub[] = { Contents of public key };

ret = wc_curve448_check_public_ex(pub, sizeof(pub), EC448_BIG_ENDIAN);
if (ret != 0) {
    // error importing key
}
```

C.16.2.18 function wc_curve448_export_public

```
int wc_curve448_export_public(
    curve448_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports a public key from the given key structure and stores the result in the out buffer. Big endian only.

Parameters:

- **key** Pointer to the curve448_key structure in from which to export the key.
- **out** Pointer to the buffer in which to store the public key.
- **outLen** On in, is the size of the out in bytes. On out, will store the bytes written to the output buffer.

See:

- wc_curve448_init

- wc_curve448_export_private_raw
- wc_curve448_import_public

Return:

- 0 Returned on successfully exporting the public key from the curve448_key structure.
- ECC_BAD_ARG_E Returned if outLen is less than CURVE448_PUB_KEY_SIZE, or if neither the public nor the private key has been set.
- BAD_FUNC_ARG Returned if any of the input parameters are NULL.

Example

```
int ret;

byte pub[56];
int pubSz;

curve448_key key;
// initialize and make key

ret = wc_curve448_export_public(&key, pub, &pubSz);
if (ret != 0) {
    // error exporting key
}
```

C.16.2.19 function wc_curve448_export_public_ex

```
int wc_curve448_export_public_ex(
    curve448_key * key,
    byte * out,
    word32 * outLen,
    int endian
)
```

This function exports a public key from the given key structure and stores the result in the out buffer. Supports both big and little endian.

Parameters:

- **key** Pointer to the curve448_key structure in from which to export the key.
- **out** Pointer to the buffer in which to store the public key.
- **outLen** On in, is the size of the out in bytes. On out, will store the bytes written to the output buffer.
- **endian** EC448_BIG_ENDIAN or EC448_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve448_init
- wc_curve448_export_private_raw
- wc_curve448_import_public

Return:

- 0 Returned on successfully exporting the public key from the curve448_key structure.
- ECC_BAD_ARG_E Returned if outLen is less than CURVE448_PUB_KEY_SIZE, or if neither the public nor the private key has been set.
- BAD_FUNC_ARG Returned if any of the input parameters are NULL.

Example

```
int ret;

byte pub[56];
int pubSz;

curve448_key key;
// initialize and make key

ret = wc_curve448_export_public_ex(&key, pub, &pubSz, EC448_BIG_ENDIAN);
if (ret != 0) {
    // error exporting key
}
```

C.16.2.20 function wc_curve448_export_key_raw

```
int wc_curve448_export_key_raw(
    curve448_key * key,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz
)
```

This function exports a key pair from the given key structure and stores the result in the out buffer. Big endian only.

Parameters:

- **key** Pointer to the curve448_key structure in from which to export the key pair.
- **priv** Pointer to the buffer in which to store the private key.
- **privSz** On in, is the size of the priv buffer in bytes. On out, will store the bytes written to the priv buffer.
- **pub** Pointer to the buffer in which to store the public key.
- **pubSz** On in, is the size of the pub buffer in bytes. On out, will store the bytes written to the pub buffer.

See:

- wc_curve448_export_key_raw_ex
- wc_curve448_export_private_raw

Return:

- 0 Returned on successfully exporting the key pair from the curve448_key structure.
- BAD_FUNC_ARG Returned if any input parameters are NULL.
- ECC_BAD_ARG_E Returned if privSz is less than CURVE448_KEY_SIZE or pubSz is less than CURVE448_PUB_KEY_SIZE.

Example

```
int ret;

byte pub[56];
byte priv[56];
int pubSz;
int privSz;

curve448_key key;
// initialize and make key
```

```
ret = wc_curve448_export_key_raw(&key, priv, &privSz, pub, &pubSz);
if (ret != 0) {
    // error exporting key
}
```

C.16.2.21 function wc_curve448_export_key_raw_ex

```
int wc_curve448_export_key_raw_ex(
    curve448_key * key,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz,
    int endian
)
```

Export curve448 key pair. Big or little endian.

Parameters:

- **key** Pointer to the curve448_key structure in from which to export the key pair.
- **priv** Pointer to the buffer in which to store the private key.
- **privSz** On in, is the size of the priv buffer in bytes. On out, will store the bytes written to the priv buffer.
- **pub** Pointer to the buffer in which to store the public key.
- **pubSz** On in, is the size of the pub buffer in bytes. On out, will store the bytes written to the pub buffer.
- **endian** EC448_BIG_ENDIAN or EC448_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve448_export_key_raw
- wc_curve448_export_private_raw_ex
- wc_curve448_export_public_ex

Return:

- 0 Success
- BAD_FUNC_ARG Returned if any input parameters are NULL.
- ECC_BAD_ARG_E Returned if privSz is less than CURVE448_KEY_SIZE or pubSz is less than CURVE448_PUB_KEY_SIZE.

This function exports a key pair from the given key structure and stores the result in the out buffer. Big or little endian.

Example

```
int ret;

byte pub[56];
byte priv[56];
int pubSz;
int privSz;

curve448_key key;
// initialize and make key

ret = wc_curve448_export_key_raw_ex(&key,priv, &privSz, pub, &pubSz,
```

```
        EC448_BIG_ENDIAN);
if (ret != 0) {
    // error exporting key
}
```

C.16.2.22 function wc_curve448_size

```
int wc_curve448_size(
    curve448_key * key
)
```

This function returns the key size of the given key structure.

Parameters:

- **key** Pointer to the curve448_key structure in for which to determine the key size.

See:

- wc_curve448_init
- wc_curve448_make_key

Return:

- Success Given a valid, initialized curve448_key structure, returns the size of the key.
- 0 Returned if key is NULL.

Example

```
int keySz;

curve448_key key;
// initialize and make key

keySz = wc_curve448_size(&key);
```

C.16.2.23 function wc_curve448_make_pub

```
int wc_curve448_make_pub(
    int public_size,
    byte * pub,
    int private_size,
    const byte * priv
)
```

This function generates a Curve448 public key from a given private key. It computes the public key by performing scalar multiplication of the base point with the private key.

Parameters:

- **public_size** size of the public key buffer (must be 56 bytes)
- **pub** pointer to buffer to store the generated public key
- **private_size** size of the private key (must be 56 bytes)
- **priv** pointer to the private key buffer

See:

- wc_curve448_make_key
- wc_curve448_import_private

Return:

- 0 On success.
- ECC_BAD_ARG_E If public_size is not CURVE448_PUB_KEY_SIZE or if private_size is not CURVE448_KEY_SIZE.
- BAD_FUNC_ARG If pub or priv is NULL.

Example

```
byte priv[CURVE448_KEY_SIZE] = { }; // private key
byte pub[CURVE448_PUB_KEY_SIZE];

int ret = wc_curve448_make_pub(CURVE448_PUB_KEY_SIZE, pub,
                               CURVE448_KEY_SIZE, priv);
if (ret != 0) {
    // error generating public key
}
```

C.16.2.24 function wc_curve448_generic

```
int wc_curve448_generic(
    int public_size,
    byte * pub,
    int private_size,
    const byte * priv,
    int basepoint_size,
    const byte * basepoint
)
```

This function performs a generic Curve448 scalar multiplication with a custom basepoint. This allows computing scalar * basepoint for any basepoint, not just the standard generator. As in wc_curve448_shared_secret_ex, an all-zero result is rejected (RFC 7748 section 6.2); build with WOLFSSL_NO_ECDHX_SHARED_ZERO_CHECK to get the raw scalar multiplication result instead.

Parameters:

- **public_size** Size of the output buffer (must be 56)
- **pub** Pointer to buffer to store the result
- **private_size** Size of the scalar (must be 56)
- **priv** Pointer to buffer containing the scalar
- **basepoint_size** Size of the basepoint (must be 56)
- **basepoint** Pointer to buffer containing the basepoint

See:

- wc_curve448_make_pub
- wc_curve448_shared_secret

Return:

- 0 On successfully computing the result
- ECC_BAD_ARG_E If any input parameter is NULL or a size is invalid
- ECC_OUT_OF_RANGE_E If the result is all-zero, i.e. the basepoint was of small order

Example

```
byte scalar[CURVE448_KEY_SIZE];
byte basepoint[CURVE448_KEY_SIZE];
byte result[CURVE448_PUB_KEY_SIZE];

// initialize scalar and basepoint
int ret = wc_curve448_generic(sizeof(result), result,
```

```
                                sizeof(scalar), scalar,
                                sizeof(basepoint), basepoint);
if (ret != 0) {
    // error computing result
}
```

C.16.3 Source code

```
int wc_curve448_make_key(WC_RNG* rng, int keysize, curve448_key* key);

int wc_curve448_shared_secret(curve448_key* private_key,
                              curve448_key* public_key,
                              byte* out, word32* outlen);

int wc_curve448_shared_secret_ex(curve448_key* private_key,
                                 curve448_key* public_key,
                                 byte* out, word32* outlen, int endian);

int wc_curve448_init(curve448_key* key);

int wc_curve448_init_ex(curve448_key* key, void* heap, int devId);

curve448_key* wc_curve448_new(void* heap, int devId, int* result_code);

int wc_curve448_delete(curve448_key* key, curve448_key** key_p);

void wc_curve448_free(curve448_key* key);

int wc_curve448_import_private(const byte* priv, word32 privSz,
                               curve448_key* key);

int wc_curve448_import_private_ex(const byte* priv, word32 privSz,
                                  curve448_key* key, int endian);

int wc_curve448_import_private_raw(const byte* priv, word32 privSz,
                           const byte* pub, word32 pubSz, curve448_key* key);

int wc_curve448_import_private_raw_ex(const byte* priv, word32 privSz,
                                      const byte* pub, word32 pubSz,
                                      curve448_key* key, int endian);

int wc_curve448_export_private_raw(curve448_key* key, byte* out,
                                   word32* outLen);

int wc_curve448_export_private_raw_ex(curve448_key* key, byte* out,
                                      word32* outLen, int endian);

int wc_curve448_import_public(const byte* in, word32 inLen,
                              curve448_key* key);

int wc_curve448_import_public_ex(const byte* in, word32 inLen,
                                 curve448_key* key, int endian);
```

```
int wc_curve448_check_public(const byte* pub, word32 pubSz, int endian);

int wc_curve448_export_public(curve448_key* key, byte* out, word32* outLen);

int wc_curve448_export_public_ex(curve448_key* key, byte* out,
                                 word32* outLen, int endian);

int wc_curve448_export_key_raw(curve448_key* key,
                               byte* priv, word32 *privSz,
                               byte* pub, word32 *pubSz);

int wc_curve448_export_key_raw_ex(curve448_key* key,
                                  byte* priv, word32 *privSz,
                                  byte* pub, word32 *pubSz,
                                  int endian);

int wc_curve448_size(curve448_key* key);

int wc_curve448_make_pub(int public_size, byte* pub, int private_size,
                         const byte* priv);

int wc_curve448_generic(int public_size, byte* pub, int private_size,
                        const byte* priv, int basepoint_size,
                        const byte* basepoint);
```

C.17 des3.h

C.17.1 Functions

	Name
int	**wc_Des_SetKey**(Des * des, const byte * key, const byte * iv, int dir)This function sets the key and initialization vector (iv) for the Des structure given as argument. It also initializes and allocates space for the buffers needed for encryption and decryption, if these have not yet been initialized. Note: If no iv is provided (i.e. iv == NULL) the initialization vector defaults to an iv of 0.
void	**wc_Des_SetIV**(Des * des, const byte * iv)This function sets the initialization vector (iv) for the Des structure given as argument. When passed a NULL iv, it sets the initialization vector to 0.
int	**wc_Des_CbcEncrypt**(Des * des, byte * out, const byte * in, word32 sz)This function encrypts the input message, in, and stores the result in the output buffer, out. It uses DES encryption with cipher block chaining (CBC) mode.

	Name
int	**wc_Des_CbcDecrypt**(Des * des, byte * out, const byte * in, word32 sz)This function decrypts the input ciphertext, in, and stores the resulting plaintext in the output buffer, out. It uses DES encryption with cipher block chaining (CBC) mode.
int	**wc_Des_EcbEncrypt**(Des * des, byte * out, const byte * in, word32 sz)This function encrypts the input message, in, and stores the result in the output buffer, out. It uses Des encryption with Electronic Codebook (ECB) mode.
int	**wc_Des3_EcbEncrypt**(Des3 * des, byte * out, const byte * in, word32 sz)This function encrypts the input message, in, and stores the result in the output buffer, out. It uses Des3 encryption with Electronic Codebook (ECB) mode. Warning: In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB API's directly whenever possible.
int	**wc_Des3_SetKey**(Des3 * des, const byte * key, const byte * iv, int dir)This function sets the key and initialization vector (iv) for the Des3 structure given as argument. It also initializes and allocates space for the buffers needed for encryption and decryption, if these have not yet been initialized. Note: If no iv is provided (i.e. iv == NULL) the initialization vector defaults to an iv of 0.
int	**wc_Des3_SetIV**(Des3 * des, const byte * iv)This function sets the initialization vector (iv) for the Des3 structure given as argument. When passed a NULL iv, it sets the initialization vector to 0.
int	**wc_Des3_CbcEncrypt**(Des3 * des, byte * out, const byte * in, word32 sz)This function encrypts the input message, in, and stores the result in the output buffer, out. It uses Triple Des (3DES) encryption with cipher block chaining (CBC) mode.
int	**wc_Des3_CbcDecrypt**(Des3 * des, byte * out, const byte * in, word32 sz)This function decrypts the input ciphertext, in, and stores the resulting plaintext in the output buffer, out. It uses Triple Des (3DES) encryption with cipher block chaining (CBC) mode.

	Name
int	**wc_Des_EcbDecrypt**(Des * des, byte * out, const byte * in, word32 sz)This function decrypts the input ciphertext and stores the resulting plaintext in the output buffer. It uses DES encryption with Electronic Codebook (ECB) mode. Warning: In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB APIs directly whenever possible.
int	**wc_Des3_EcbDecrypt**(Des3 * des, byte * out, const byte * in, word32 sz)This function decrypts the input ciphertext and stores the resulting plaintext in the output buffer. It uses Triple DES (3DES) encryption with Electronic Codebook (ECB) mode. Warning: In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB APIs directly whenever possible.
int	**wc_Des3Init**(Des3 * des3, void * heap, int devId)This function initializes a Des3 structure for use with hardware acceleration and custom memory management. This is an extended version of the standard initialization that allows specification of heap hints and device IDs.
void	**wc_Des3Free**(Des3 * des3)This function frees a Des3 structure and releases any resources allocated for it. This should be called when finished using the Des3 structure to prevent memory leaks.

C.17.2 Functions Documentation

C.17.2.1 function wc_Des_SetKey

```
int wc_Des_SetKey(
    Des * des,
    const byte * key,
    const byte * iv,
    int dir
)
```

This function sets the key and initialization vector (iv) for the Des structure given as argument. It also initializes and allocates space for the buffers needed for encryption and decryption, if these have not yet been initialized. Note: If no iv is provided (i.e. iv == NULL) the initialization vector defaults to an iv of 0.

Parameters:

- **des** pointer to the Des structure to initialize
- **key** pointer to the buffer containing the 8 byte key with which to initialize the Des structure
- **iv** pointer to the buffer containing the 8 byte iv with which to initialize the Des structure. If this is not provided, the iv defaults to 0
- **dir** direction of encryption. Valid options are: DES_ENCRYPTION, and DES_DECRYPTION

See:

- wc_Des_SetIV
- wc_Des3_SetKey

Return: 0 On successfully setting the key and initialization vector for the Des structure

Note: This function must be called on a Des structure before it is passed to wc_Des_CbcEncrypt(), wc_Des_CbcDecrypt(), wc_Des_EcbEncrypt(), or wc_Des_EcbDecrypt(). Unlike the Des3 API, the Des structure has no key-set state, so those functions do not verify that a key has been set and will operate on undefined key material if this precondition is not met.

3

Example

```
Des enc; // Des structure used for encryption
int ret;
byte key[] = { // initialize with 8 byte key };
byte iv[]  = { // initialize with 8 byte iv };

ret = wc_Des_SetKey(&des, key, iv, DES_ENCRYPTION);
if (ret != 0) {
    // error initializing des structure
}
```

C.17.2.2 function wc_Des_SetIV

```
void wc_Des_SetIV(
    Des * des,
    const byte * iv
)
```

This function sets the initialization vector (iv) for the Des structure given as argument. When passed a NULL iv, it sets the initialization vector to 0.

Parameters:

- **des** pointer to the Des structure for which to set the iv
- **iv** pointer to the buffer containing the 8 byte iv with which to initialize the Des structure. If this is not provided, the iv defaults to 0

See: wc_Des_SetKey

Return: none No returns.

3

Example

```
Des enc; // Des structure used for encryption
// initialize enc with wc_Des_SetKey
byte iv[]  = { // initialize with 8 byte iv };
wc_Des_SetIV(&enc, iv);
}
```

C.17.2.3 function wc_Des_CbcEncrypt

```
int wc_Des_CbcEncrypt(
    Des * des,
    byte * out,
```

```
    const byte * in,
    word32 sz
)
```

This function encrypts the input message, in, and stores the result in the output buffer, out. It uses DES encryption with cipher block chaining (CBC) mode.

Parameters:

- **des** pointer to the Des structure to use for encryption
- **out** pointer to the buffer in which to store the encrypted ciphertext
- **in** pointer to the input buffer containing the message to encrypt
- **sz** length of the message to encrypt

See:

- wc_Des_SetKey
- wc_Des_CbcDecrypt

Return: 0 Returned upon successfully encrypting the given input message

Note: wc_Des_SetKey() must be called on des before calling this function. This function does not check whether a key has been set and will encrypt using undefined key material if it has not.

3

Example

```
Des enc; // Des structure used for encryption
// initialize enc with wc_Des_SetKey, use mode DES_ENCRYPTION

byte plain[]  = { // initialize with message };
byte cipher[sizeof(plain)];

if ( wc_Des_CbcEncrypt(&enc, cipher, plain, sizeof(plain)) != 0) {
    // error encrypting message
}
```

C.17.2.4 function wc_Des_CbcDecrypt

```
int wc_Des_CbcDecrypt(
    Des * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function decrypts the input ciphertext, in, and stores the resulting plaintext in the output buffer, out. It uses DES encryption with cipher block chaining (CBC) mode.

Parameters:

- **des** pointer to the Des structure to use for decryption
- **out** pointer to the buffer in which to store the decrypted plaintext
- **in** pointer to the input buffer containing the encrypted ciphertext
- **sz** length of the ciphertext to decrypt

See:

- wc_Des_SetKey
- wc_Des_CbcEncrypt

Return: 0 Returned upon successfully decrypting the given ciphertext

Note: wc_Des_SetKey() must be called on des before calling this function. This function does not check whether a key has been set and will decrypt using undefined key material if it has not.

3

Example

```
Des dec; // Des structure used for decryption
// initialize dec with wc_Des_SetKey, use mode DES_DECRYPTION

byte cipher[]  = { // initialize with ciphertext };
byte decoded[sizeof(cipher)];

if ( wc_Des_CbcDecrypt(&dec, decoded, cipher, sizeof(cipher)) != 0) {
    // error decrypting message
}
```

C.17.2.5 function wc_Des_EcbEncrypt

```
int wc_Des_EcbEncrypt(
    Des * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function encrypts the input message, in, and stores the result in the output buffer, out. It uses Des encryption with Electronic Codebook (ECB) mode.

Parameters:

- **des** pointer to the Des structure to use for encryption
- **out** pointer to the buffer in which to store the encrypted message
- **in** pointer to the input buffer containing the plaintext to encrypt
- **sz** length of the plaintext to encrypt

See: wc_Des_SetKey

Return: 0: Returned upon successfully encrypting the given plaintext.

Note: wc_Des_SetKey() must be called on des before calling this function. This function does not check whether a key has been set and will encrypt using undefined key material if it has not.

3

Example

```
Des enc; // Des structure used for encryption
// initialize enc with wc_Des_SetKey, use mode DES_ENCRYPTION

byte plain[]  = { // initialize with message to encrypt };
byte cipher[sizeof(plain)];

if ( wc_Des_EcbEncrypt(&enc,cipher, plain, sizeof(plain)) != 0) {
    // error encrypting message
}
```

C.17.2.6 function wc_Des3_EcbEncrypt

```
int wc_Des3_EcbEncrypt(
    Des3 * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function encrypts the input message, in, and stores the result in the output buffer, out. It uses Des3 encryption with Electronic Codebook (ECB) mode. Warning: In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB API's directly whenever possible.

Parameters:

- **des** pointer to the Des3 structure to use for encryption
- **out** pointer to the buffer in which to store the encrypted message
- **in** pointer to the input buffer containing the plaintext to encrypt
- **sz** length of the plaintext to encrypt

See: wc_Des3_SetKey

Return:

- 0 Returned upon successfully encrypting the given plaintext
- BAD_FUNC_ARG If des, out, or in is NULL.
- MISSING_KEY If no key has been set on the Des3 structure with wc_Des3_SetKey.

3

Example

```
Des3 enc; // Des3 structure used for encryption
// initialize enc with wc_Des3_SetKey, use mode DES_ENCRYPTION

byte plain[]  = { // initialize with message to encrypt };
byte cipher[sizeof(plain)];

if ( wc_Des3_EcbEncrypt(&enc,cipher, plain, sizeof(plain)) != 0) {
    // error encrypting message
}
```

C.17.2.7 function wc_Des3_SetKey

```
int wc_Des3_SetKey(
    Des3 * des,
    const byte * key,
    const byte * iv,
    int dir
)
```

This function sets the key and initialization vector (iv) for the Des3 structure given as argument. It also initializes and allocates space for the buffers needed for encryption and decryption, if these have not yet been initialized. Note: If no iv is provided (i.e. iv == NULL) the initialization vector defaults to an iv of 0.

Parameters:

- **des3** pointer to the Des3 structure to initialize
- **key** pointer to the buffer containing the 24 byte key with which to initialize the Des3 structure

- **iv** pointer to the buffer containing the 8 byte iv with which to initialize the Des3 structure. If this is not provided, the iv defaults to 0
- **dir** direction of encryption. Valid options are: DES_ENCRYPTION, and DES_DECRYPTION

See:

- wc_Des3_SetIV
- wc_Des3_CbcEncrypt
- wc_Des3_CbcDecrypt

Return: 0 On successfully setting the key and initialization vector for the Des structure

3

Example

```
Des3 enc; // Des3 structure used for encryption
int ret;
byte key[] = { // initialize with 24 byte key };
byte iv[]  = { // initialize with 8 byte iv };

ret = wc_Des3_SetKey(&des, key, iv, DES_ENCRYPTION);
if (ret != 0) {
    // error initializing des structure
}
```

C.17.2.8 function wc_Des3_SetIV

```
int wc_Des3_SetIV(
    Des3 * des,
    const byte * iv
)
```

This function sets the initialization vector (iv) for the Des3 structure given as argument. When passed a NULL iv, it sets the initialization vector to 0.

Parameters:

- **des** pointer to the Des3 structure for which to set the iv
- **iv** pointer to the buffer containing the 8 byte iv with which to initialize the Des3 structure. If this is not provided, the iv defaults to 0

See: wc_Des3_SetKey

Return: none No returns.

3

Example

```
Des3 enc; // Des3 structure used for encryption
// initialize enc with wc_Des3_SetKey

byte iv[]  = { // initialize with 8 byte iv };

wc_Des3_SetIV(&enc, iv);
}
```

C.17.2.9 function wc_Des3_CbcEncrypt

```
int wc_Des3_CbcEncrypt(
    Des3 * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function encrypts the input message, in, and stores the result in the output buffer, out. It uses Triple Des (3DES) encryption with cipher block chaining (CBC) mode.

Parameters:

- **des** pointer to the Des3 structure to use for encryption
- **out** pointer to the buffer in which to store the encrypted ciphertext
- **in** pointer to the input buffer containing the message to encrypt
- **sz** length of the message to encrypt

See:

- wc_Des3_SetKey
- wc_Des3_CbcDecrypt

Return:

- 0 Returned upon successfully encrypting the given input message
- BAD_FUNC_ARG If des, out, or in is NULL.
- BAD_LENGTH_E If sz is not a multiple of DES_BLOCK_SIZE.
- MISSING_KEY If no key has been set on the Des3 structure with wc_Des3_SetKey.

3

Example

```
Des3 enc; // Des3 structure used for encryption
// initialize enc with wc_Des3_SetKey, use mode DES_ENCRYPTION

byte plain[]  = { // initialize with message };
byte cipher[sizeof(plain)];

if ( wc_Des3_CbcEncrypt(&enc, cipher, plain, sizeof(plain)) != 0) {
    // error encrypting message
}
```

C.17.2.10 function wc_Des3_CbcDecrypt

```
int wc_Des3_CbcDecrypt(
    Des3 * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function decrypts the input ciphertext, in, and stores the resulting plaintext in the output buffer, out. It uses Triple Des (3DES) encryption with cipher block chaining (CBC) mode.

Parameters:

- **des** pointer to the Des3 structure to use for decryption
- **out** pointer to the buffer in which to store the decrypted plaintext
- **in** pointer to the input buffer containing the encrypted ciphertext
- **sz** length of the ciphertext to decrypt

See:

- wc_Des3_SetKey
- wc_Des3_CbcEncrypt

Return:

- 0 Returned upon successfully decrypting the given ciphertext
- BAD_FUNC_ARG If des, out, or in is NULL.
- BAD_LENGTH_E If sz is not a multiple of DES_BLOCK_SIZE.
- MISSING_KEY If no key has been set on the Des3 structure with wc_Des3_SetKey.

3

Example

```
Des3 dec; // Des structure used for decryption
// initialize dec with wc_Des3_SetKey, use mode DES_DECRYPTION

byte cipher[]  = { // initialize with ciphertext };
byte decoded[sizeof(cipher)];

if ( wc_Des3_CbcDecrypt(&dec, decoded, cipher, sizeof(cipher)) != 0) {
    // error decrypting message
}
```

C.17.2.11 function wc_Des_EcbDecrypt

```
int wc_Des_EcbDecrypt(
    Des * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function decrypts the input ciphertext and stores the resulting plaintext in the output buffer. It uses DES encryption with Electronic Codebook (ECB) mode. Warning: In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB APIs directly whenever possible.

Parameters:

- **des** pointer to the Des structure to use for decryption
- **out** pointer to the buffer in which to store the decrypted plaintext
- **in** pointer to the input buffer containing the ciphertext
- **sz** length of the ciphertext to decrypt

See:

- wc_Des_SetKey
- wc_Des_EcbEncrypt

Return: 0 On successfully decrypting the given ciphertext

Note: wc_Des_SetKey() must be called on des before calling this function. This function does not check whether a key has been set and will decrypt using undefined key material if it has not.

3

Example

```
Des dec;
byte cipher[]; // ciphertext to decrypt
```

```
byte plain[sizeof(cipher)];

wc_Des_SetKey(&dec, key, iv, DES_DECRYPTION);
if (wc_Des_EcbDecrypt(&dec, plain, cipher, sizeof(cipher)) != 0) {
    // error decrypting message
}
```

C.17.2.12 function wc_Des3_EcbDecrypt

```
int wc_Des3_EcbDecrypt(
    Des3 * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function decrypts the input ciphertext and stores the resulting plaintext in the output buffer. It uses Triple DES (3DES) encryption with Electronic Codebook (ECB) mode. Warning: In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB APIs directly whenever possible.

Parameters:

- **des** pointer to the Des3 structure to use for decryption
- **out** pointer to the buffer in which to store the decrypted plaintext
- **in** pointer to the input buffer containing the ciphertext
- **sz** length of the ciphertext to decrypt

See:

- wc_Des3_SetKey
- wc_Des3_EcbEncrypt

Return:

- 0 On successfully decrypting the given ciphertext
- BAD_FUNC_ARG If des, out, or in is NULL.
- MISSING_KEY If no key has been set on the Des3 structure with wc_Des3_SetKey.

3

Example

```
Des3 dec;
byte cipher[]; // ciphertext to decrypt
byte plain[sizeof(cipher)];

wc_Des3_SetKey(&dec, key, iv, DES_DECRYPTION);
if (wc_Des3_EcbDecrypt(&dec, plain, cipher, sizeof(cipher)) != 0) {
    // error decrypting message
}
```

C.17.2.13 function wc_Des3Init

```
int wc_Des3Init(
    Des3 * des3,
    void * heap,
    int devId
)
```

This function initializes a Des3 structure for use with hardware acceleration and custom memory management. This is an extended version of the standard initialization that allows specification of heap hints and device IDs.

Parameters:

- **des3** pointer to the Des3 structure to initialize
- **heap** pointer to heap hint for memory allocation (can be NULL)
- **devId** device ID for hardware acceleration (use INVALID_DEVID for software only)

See:

- wc_Des3_SetKey
- wc_Des3Free

Return:

- 0 On successfully initializing the Des3 structure
- BAD_FUNC_ARG If des3 is NULL

3

Example

```
Des3 des;
void* heap = NULL;
int devId = INVALID_DEVID;

if (wc_Des3Init(&des, heap, devId) != 0) {
    // error initializing Des3 structure
}
```

C.17.2.14 function wc_Des3Free

```
void wc_Des3Free(
    Des3 * des3
)
```

This function frees a Des3 structure and releases any resources allocated for it. This should be called when finished using the Des3 structure to prevent memory leaks.

Parameters:

- **des3** pointer to the Des3 structure to free

See:

- wc_Des3Init
- wc_Des3_SetKey

Return: none No returns.

3

Example

```
Des3 des;
wc_Des3Init(&des, NULL, INVALID_DEVID);
wc_Des3_SetKey(&des, key, iv, DES_ENCRYPTION);
// use des for encryption/decryption
wc_Des3Free(&des);
```

C.17.3 Source code

```
int  wc_Des_SetKey(Des* des, const byte* key,
                                const byte* iv, int dir);

void wc_Des_SetIV(Des* des, const byte* iv);

int  wc_Des_CbcEncrypt(Des* des, byte* out,
                                  const byte* in, word32 sz);

int  wc_Des_CbcDecrypt(Des* des, byte* out,
                                  const byte* in, word32 sz);

int  wc_Des_EcbEncrypt(Des* des, byte* out,
                                  const byte* in, word32 sz);

int wc_Des3_EcbEncrypt(Des3* des, byte* out,
                                  const byte* in, word32 sz);

int  wc_Des3_SetKey(Des3* des, const byte* key,
                                const byte* iv,int dir);

int  wc_Des3_SetIV(Des3* des, const byte* iv);

int  wc_Des3_CbcEncrypt(Des3* des, byte* out,
                                   const byte* in,word32 sz);

int  wc_Des3_CbcDecrypt(Des3* des, byte* out,
                                   const byte* in,word32 sz);

int wc_Des_EcbDecrypt(Des* des, byte* out, const byte* in, word32 sz);

int wc_Des3_EcbDecrypt(Des3* des, byte* out, const byte* in, word32 sz);

int wc_Des3Init(Des3* des3, void* heap, int devId);

void wc_Des3Free(Des3* des3);
```

C.18 dh.h

C.18.1 Functions

	Name
int	**wc_InitDhKey**(DhKey * key)This function initializes a Diffie-Hellman key for use in negotiating a secure secret key with the Diffie-Hellman exchange protocol.
int	**wc_FreeDhKey**(DhKey * key)This function frees a Diffie-Hellman key after it has been used to negotiate a secure secret key with the Diffie-Hellman exchange protocol.

	Name
int	**wc_DhGenerateKeyPair**(DhKey * key, WC_RNG * rng, byte * priv, word32 * privSz, byte * pub, word32 * pubSz)This function generates a public/private key pair based on the Diffie-Hellman public parameters, storing the private key in priv and the public key in pub. It takes an initialized Diffie-Hellman key and an initialized rng structure.
int	**wc_DhAgree**(DhKey * key, byte * agree, word32 * agreeSz, const byte * priv, word32 privSz, const byte * otherPub, word32 pubSz)This function generates an agreed upon secret key based on a local private key and a received public key. If completed on both sides of an exchange, this function generates an agreed upon secret key for symmetric communication. On successfully generating a shared secret key, the size of the secret key written will be stored in agreeSz.
int	**wc_DhKeyDecode**(const byte * input, word32 * inOutIdx, DhKey * key, word32 inSz)This function decodes a Diffie-Hellman key from the given input buffer containing the key in DER format. It stores the result in the DhKey structure.
int	**wc_DhSetKey**(DhKey * key, const byte * p, word32 pSz, const byte * g, word32 gSz)This function sets the key for a DhKey structure using the input private key parameters. Unlike wc_DhKeyDecode, this function does not require that the input key be formatted in DER format, and instead simply accepts the parsed input parameters p (prime) and g (base).
int	**wc_DhParamsLoad**(const byte * input, word32 inSz, byte * p, word32 * pInOutSz, byte * g, word32 * gInOutSz)This function loads the Diffie_Hellman parameters, p (prime) and g (base) out of the given input buffer, DER formatted.
int	**wolfSSL_i2d_DHparams**(const WOLFSSL_DH * dh, unsigned char ** out)Encodes DH parameters to DER format for OpenSSL compatibility.
WOLFSSL_DH *	**wolfSSL_DH_new**(void)Allocates and initializes a new DH structure for OpenSSL compatibility.
WOLFSSL_DH *	**wolfSSL_DH_new_by_nid**(int nid)Creates a new DH structure with named group parameters.
void	**wolfSSL_DH_free**(WOLFSSL_DH * dh)Frees a DH structure.
WOLFSSL_DH *	**wolfSSL_DH_dup**(WOLFSSL_DH * dh)Duplicates a DH structure.

	Name
int	**wolfSSL_DH_up_ref**(WOLFSSL_DH * dh)Increments reference count for DH structure.
int	**wolfSSL_DH_check**(const WOLFSSL_DH * dh, int * codes)Validates DH parameters.
int	**wolfSSL_DH_size**(WOLFSSL_DH * dh)Returns size of DH key in bytes.
int	**wolfSSL_DH_generate_key**(WOLFSSL_DH * dh)Generates DH public/private key pair.
int	**wolfSSL_DH_compute_key**(unsigned char * key, const WOLFSSL_BIGNUM * pub, WOLFSSL_DH * dh)Computes shared secret from peer's public key.
int	**wolfSSL_DH_compute_key_padded**(unsigned char * key, const WOLFSSL_BIGNUM * otherPub, WOLFSSL_DH * dh)Computes shared secret with zero-padding to DH size.
int	**wolfSSL_DH_LoadDer**(WOLFSSL_DH * dh, const unsigned char * derBuf, int derSz)Loads DH parameters from DER buffer.
int	**wolfSSL_DH_set_length**(WOLFSSL_DH * dh, long len)Sets optional private key length.
int	**wolfSSL_DH_set0_pqg**(WOLFSSL_DH * dh, WOLFSSL_BIGNUM * p, WOLFSSL_BIGNUM * q, WOLFSSL_BIGNUM * g)Sets DH parameters p, q, and g.
WOLFSSL_DH *	**wolfSSL_DH_get_2048_256**(void)Returns DH parameters for 2048-bit MODP group with 256-bit subgroup.
const DhParams *	**wc_Dh_ffdhe2048_Get**(void)Returns FFDHE 2048-bit group parameters.
const DhParams *	**wc_Dh_ffdhe3072_Get**(void)Returns FFDHE 3072-bit group parameters.
const DhParams *	**wc_Dh_ffdhe4096_Get**(void)Returns FFDHE 4096-bit group parameters.
const DhParams *	**wc_Dh_ffdhe6144_Get**(void)Returns FFDHE 6144-bit group parameters.
const DhParams *	**wc_Dh_ffdhe8192_Get**(void)Returns FFDHE 8192-bit group parameters.
int	**wc_InitDhKey_ex**(DhKey * key, void * heap, int devId)Initializes DH key with heap hint and device ID.
int	**wc_DhAgree_ct**(DhKey * key, byte * agree, word32 * agreeSz, const byte * priv, word32 privSz, const byte * otherPub, word32 pubSz)Computes shared secret with constant-time operations.
int	**wc_DhSetNamedKey**(DhKey * key, int name)Sets DH key to use named group parameters.

	Name
int	**wc_DhGetNamedKeyParamSize**(int name, word32 * p, word32 * g, word32 * q)Gets parameter sizes for named group.
word32	**wc_DhGetNamedKeyMinSize**(int name)Gets minimum key size for named group.
int	**wc_DhCmpNamedKey**(int name, int noQ, const byte * p, word32 pSz, const byte * g, word32 gSz, const byte * q, word32 qSz)Compares parameters against named group.
int	**wc_DhCopyNamedKey**(int name, byte * p, word32 * pSz, byte * g, word32 * gSz, byte * q, word32 * qSz)Copies named group parameters to buffers.
int	**wc_DhGeneratePublic**(DhKey * key, byte * priv, word32 privSz, byte * pub, word32 * pubSz)Generates public key from private key.
int	**wc_DhImportKeyPair**(DhKey * key, const byte * priv, word32 privSz, const byte * pub, word32 pubSz)Imports private and/or public key into DH key.
int	**wc_DhExportKeyPair**(DhKey * key, byte * priv, word32 * pPrivSz, byte * pub, word32 * pPubSz)Exports private and public key from DH key.
int	**wc_DhCheckPubValue**(const byte * prime, word32 primeSz, const byte * pub, word32 pubSz)Validates public key value.
int	**wc_DhCheckKeyPair**(DhKey * key, const byte * pub, word32 pubSz, const byte * priv, word32 privSz)Checks DH keys for pair_wise consistency per process in SP 800_56Ar3, section 5.6.2.1.4, method (b) for FFC.
int	**wc_DhCheckPrivKey**(DhKey * key, const byte * priv, word32 pubSz)Check DH private key for invalid numbers.
int	**wc_DhCheckPrivKey_ex**(DhKey * key, const byte * priv, word32 pubSz, const byte * prime, word32 primeSz)
int	**wc_DhCheckPubKey**(DhKey * key, const byte * pub, word32 pubSz)
int	**wc_DhCheckPubKey_ex**(DhKey * key, const byte * pub, word32 pubSz, const byte * prime, word32 primeSz)
int	**wc_DhExportParamsRaw**(DhKey * dh, byte * p, word32 * pSz, byte * q, word32 * qSz, byte * g, word32 * gSz)
int	**wc_DhGenerateParams**(WC_RNG * rng, int modSz, DhKey * dh)

	Name
int	**wc_DhSetCheckKey**(DhKey * key, const byte * p, word32 pSz, const byte * g, word32 gSz, const byte * q, word32 qSz, int trusted, WC_RNG * rng)
int	**wc_DhSetKey_ex**(DhKey * key, const byte * p, word32 pSz, const byte * g, word32 gSz, const byte * q, word32 qSz)

C.18.2 Functions Documentation

C.18.2.1 function wc_InitDhKey

```
int wc_InitDhKey(
    DhKey * key
)
```

This function initializes a Diffie-Hellman key for use in negotiating a secure secret key with the Diffie-Hellman exchange protocol.

Parameters:

- **key** pointer to the DhKey structure to initialize for use with secure key exchanges

See:

- wc_FreeDhKey
- wc_DhGenerateKeyPair

Return: none No returns.

Example

```
DhKey key;
wc_InitDhKey(&key); // initialize DH key
```

C.18.2.2 function wc_FreeDhKey

```
int wc_FreeDhKey(
    DhKey * key
)
```

This function frees a Diffie-Hellman key after it has been used to negotiate a secure secret key with the Diffie-Hellman exchange protocol.

Parameters:

- **key** pointer to the DhKey structure to free

See: wc_InitDhKey

Return: none No returns.

Example

```
DhKey key;
// initialize key, perform key exchange

wc_FreeDhKey(&key); // free DH key to avoid memory leaks
```

C.18.2.3 function wc_DhGenerateKeyPair

```
int wc_DhGenerateKeyPair(
    DhKey * key,
    WC_RNG * rng,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz
)
```

This function generates a public/private key pair based on the Diffie-Hellman public parameters, storing the private key in priv and the public key in pub. It takes an initialized Diffie-Hellman key and an initialized rng structure.

Parameters:

- **key** pointer to the DhKey structure from which to generate the key pair
- **rng** pointer to an initialized random number generator (rng) with which to generate the keys
- **priv** pointer to a buffer in which to store the private key
- **privSz** will store the size of the private key written to priv
- **pub** pointer to a buffer in which to store the public key
- **pubSz** will store the size of the private key written to pub

See:

- wc_InitDhKey
- wc_DhSetKey
- wc_DhKeyDecode

Return:

- BAD_FUNC_ARG Returned if there is an error parsing one of the inputs to this function
- RNG_FAILURE_E Returned if there is an error generating a random number using rng
- MP_INIT_E May be returned if there is an error in the math library while generating the public key
- MP_READ_E May be returned if there is an error in the math library while generating the public key
- MP_EXPTMOD_E May be returned if there is an error in the math library while generating the public key
- MP_TO_E May be returned if there is an error in the math library while generating the public key

Example

```
DhKey key;
int ret;
byte priv[256];
byte pub[256];
word32 privSz, pubSz;

wc_InitDhKey(&key); // initialize key
// Set DH parameters using wc_DhSetKey or wc_DhKeyDecode
WC_RNG rng;
wc_InitRng(&rng); // initialize rng
ret = wc_DhGenerateKeyPair(&key, &rng, priv, &privSz, pub, &pubSz);
```

C.18.2.4 function wc_DhAgree

```
int wc_DhAgree(
    DhKey * key,
    byte * agree,
    word32 * agreeSz,
    const byte * priv,
    word32 privSz,
    const byte * otherPub,
    word32 pubSz
)
```

This function generates an agreed upon secret key based on a local private key and a received public key. If completed on both sides of an exchange, this function generates an agreed upon secret key for symmetric communication. On successfully generating a shared secret key, the size of the secret key written will be stored in agreeSz.

Parameters:

- **key** pointer to the DhKey structure to use to compute the shared key
- **agree** pointer to the buffer in which to store the secret key
- **agreeSz** will hold the size of the secret key after successful generation
- **priv** pointer to the buffer containing the local secret key
- **privSz** size of the local secret key
- **otherPub** pointer to a buffer containing the received public key
- **pubSz** size of the received public key

See: wc_DhGenerateKeyPair

Return:

- 0 Returned on successfully generating an agreed upon secret key
- MP_INIT_E May be returned if there is an error while generating the shared secret key
- MP_READ_E May be returned if there is an error while generating the shared secret key
- MP_EXPTMOD_E May be returned if there is an error while generating the shared secret key
- MP_TO_E May be returned if there is an error while generating the shared secret key

Example

```
DhKey key;
int ret;
byte priv[256];
byte agree[256];
word32 agreeSz;

// initialize key, set key prime and base
// wc_DhGenerateKeyPair -- store private key in priv
byte pub[] = { // initialized with the received public key };
ret = wc_DhAgree(&key, agree, &agreeSz, priv, sizeof(priv), pub,
sizeof(pub));
if ( ret != 0 ) {
    // error generating shared key
}
```

C.18.2.5 function wc_DhKeyDecode

```
int wc_DhKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    DhKey * key,
```

```
    word32 inSz
)
```

This function decodes a Diffie-Hellman key from the given input buffer containing the key in DER format. It stores the result in the DhKey structure.

Parameters:

- **input** pointer to the buffer containing the DER formatted Diffie-Hellman key
- **inOutIdx** pointer to an integer in which to store the index parsed to while decoding the key
- **key** pointer to the DhKey structure to initialize with the input key
- **inSz** length of the input buffer. Gives the max length that may be read

See: wc_DhSetKey

Return:

- 0 Returned on successfully decoding the input key
- ASN_PARSE_E Returned if there is an error parsing the sequence of the input
- ASN_DH_KEY_E Returned if there is an error reading the private key parameters from the parsed input

Example

```
DhKey key;
word32 idx = 0;

byte keyBuff[1024];
// initialize with DER formatted key
wc_DhKeyInit(&key);
ret = wc_DhKeyDecode(keyBuff, &idx, &key, sizeof(keyBuff));

if ( ret != 0 ) {
    // error decoding key
}
```

C.18.2.6 function wc_DhSetKey

```
int wc_DhSetKey(
    DhKey * key,
    const byte * p,
    word32 pSz,
    const byte * g,
    word32 gSz
)
```

This function sets the key for a DhKey structure using the input private key parameters. Unlike wc_DhKeyDecode, this function does not require that the input key be formatted in DER format, and instead simply accepts the parsed input parameters p (prime) and g (base).

Parameters:

- **key** pointer to the DhKey structure on which to set the key
- **p** pointer to the buffer containing the prime for use with the key
- **pSz** length of the input prime
- **g** pointer to the buffer containing the base for use with the key
- **gSz** length of the input base

See: wc_DhKeyDecode

Return:

- 0 Returned on successfully setting the key
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL
- MP_INIT_E Returned if there is an error initializing the key parameters for storage
- ASN_DH_KEY_E Returned if there is an error reading in the DH key parameters p and g

Example

```
DhKey key;

byte p[] = { // initialize with prime };
byte g[] = { // initialize with base };
wc_DhKeyInit(&key);
ret = wc_DhSetKey(key, p, sizeof(p), g, sizeof(g));

if ( ret != 0 ) {
    // error setting key
}
```

C.18.2.7 function wc_DhParamsLoad

```
int wc_DhParamsLoad(
    const byte * input,
    word32 inSz,
    byte * p,
    word32 * pInOutSz,
    byte * g,
    word32 * gInOutSz
)
```

This function loads the Diffie-Hellman parameters, p (prime) and g (base) out of the given input buffer, DER formatted.

Parameters:

- **input** pointer to a buffer containing a DER formatted Diffie-Hellman certificate to parse
- **inSz** size of the input buffer
- **p** pointer to a buffer in which to store the parsed prime
- **pInOutSz** pointer to a word32 object containing the available size in the p buffer. Will be overwritten with the number of bytes written to the buffer after completing the function call
- **g** pointer to a buffer in which to store the parsed base
- **gInOutSz** pointer to a word32 object containing the available size in the g buffer. Will be overwritten with the number of bytes written to the buffer after completing the function call

See:

- wc_DhSetKey
- wc_DhKeyDecode

Return:

- 0 Returned on successfully extracting the DH parameters
- ASN_PARSE_E Returned if an error occurs while parsing the DER formatted DH certificate
- BUFFER_E Returned if there is inadequate space in p or g to store the parsed parameters

Example

```
byte dhCert[] = { initialize with DER formatted certificate };
byte p[MAX_DH_SIZE];
byte g[MAX_DH_SIZE];
word32 pSz = MAX_DH_SIZE;
```

```
word32 gSz = MAX_DH_SIZE;

ret = wc_DhParamsLoad(dhCert, sizeof(dhCert), p, &pSz, g, &gSz);
if ( ret != 0 ) {
    // error parsing inputs
}
```

C.18.2.8 function wolfSSL_i2d_DHparams

```
int wolfSSL_i2d_DHparams(
    const WOLFSSL_DH * dh,
    unsigned char ** out
)
```

Encodes DH parameters to DER format for OpenSSL compatibility.

Parameters:

- **dh** DH parameters to encode
- **out** Output buffer pointer (if *out is NULL, allocates buffer)

See: wolfSSL_DH_new

Return:

- Length of DER encoding on success
- Negative on error

Example

```
WOLFSSL_DH* dh = wolfSSL_DH_new();
unsigned char* der = NULL;
int derSz = wolfSSL_i2d_DHparams(dh, &der);
if (derSz > 0) {
    // use der buffer
    XFREE(der, NULL, DYNAMIC_TYPE_OPENSSL);
}
```

C.18.2.9 function wolfSSL_DH_new

```
WOLFSSL_DH * wolfSSL_DH_new(
    void
)
```

Allocates and initializes a new DH structure for OpenSSL compatibility.

See:

- wolfSSL_DH_free
- wolfSSL_DH_generate_key

Return:

- Pointer to WOLFSSL_DH on success
- NULL on failure

Example

```
WOLFSSL_DH* dh = wolfSSL_DH_new();
if (dh == NULL) {
    // error allocating DH
}
```

```
// use dh
wolfSSL_DH_free(dh);
```

C.18.2.10 function wolfSSL_DH_new_by_nid

```
WOLFSSL_DH * wolfSSL_DH_new_by_nid(
    int nid
)
```

Creates a new DH structure with named group parameters.

Parameters:

- **nid** Named group identifier (e.g., NID_ffdhe2048)

See: wolfSSL_DH_new

Return:

- Pointer to WOLFSSL_DH on success
- NULL on failure

Example

```
WOLFSSL_DH* dh = wolfSSL_DH_new_by_nid(NID_ffdhe2048);
if (dh == NULL) {
    // error creating DH with named group
}
```

C.18.2.11 function wolfSSL_DH_free

```
void wolfSSL_DH_free(
    WOLFSSL_DH * dh
)
```

Frees a DH structure.

Parameters:

- **dh** DH structure to free

See: wolfSSL_DH_new

Example

```
WOLFSSL_DH* dh = wolfSSL_DH_new();
// use dh
wolfSSL_DH_free(dh);
```

C.18.2.12 function wolfSSL_DH_dup

```
WOLFSSL_DH * wolfSSL_DH_dup(
    WOLFSSL_DH * dh
)
```

Duplicates a DH structure.

Parameters:

- **dh** DH structure to duplicate

See: wolfSSL_DH_new

Return:

- Pointer to new WOLFSSL_DH on success
- NULL on failure

Example

```
WOLFSSL_DH* dh = wolfSSL_DH_new();
WOLFSSL_DH* dhCopy = wolfSSL_DH_dup(dh);
```

C.18.2.13 function wolfSSL_DH_up_ref

```
int wolfSSL_DH_up_ref(
    WOLFSSL_DH * dh
)
```

Increments reference count for DH structure.

Parameters:

- **dh** DH structure to increment reference

See: wolfSSL_DH_free

Return:

- 1 on success
- 0 on failure

Example

```
WOLFSSL_DH* dh = wolfSSL_DH_new();
int ret = wolfSSL_DH_up_ref(dh);
```

C.18.2.14 function wolfSSL_DH_check

```
int wolfSSL_DH_check(
    const WOLFSSL_DH * dh,
    int * codes
)
```

Validates DH parameters.

Parameters:

- **dh** DH parameters to check
- **codes** Output for validation error codes

See: wolfSSL_DH_generate_key

Return:

- 1 on success
- 0 on failure

Example

```
WOLFSSL_DH* dh = wolfSSL_DH_new();
int codes;
int ret = wolfSSL_DH_check(dh, &codes);
if (ret != 1 || codes != 0) {
    // validation failed
}
```

C.18.2.15 function wolfSSL_DH_size

```
int wolfSSL_DH_size(
    WOLFSSL_DH * dh
)
```

Returns size of DH key in bytes.

Parameters:

- **dh** DH structure

See: wolfSSL_DH_new

Return:

- Key size in bytes on success
- -1 on failure

Example

```
WOLFSSL_DH* dh = wolfSSL_DH_new();
int size = wolfSSL_DH_size(dh);
```

C.18.2.16 function wolfSSL_DH_generate_key

```
int wolfSSL_DH_generate_key(
    WOLFSSL_DH * dh
)
```

Generates DH public/private key pair.

Parameters:

- **dh** DH structure with parameters set

See: wolfSSL_DH_compute_key

Return:

- 1 on success
- 0 on failure

Example

```
WOLFSSL_DH* dh = wolfSSL_DH_new();
// set p and g parameters
int ret = wolfSSL_DH_generate_key(dh);
if (ret != 1) {
    // key generation failed
}
```

C.18.2.17 function wolfSSL_DH_compute_key

```
int wolfSSL_DH_compute_key(
    unsigned char * key,
    const WOLFSSL_BIGNUM * pub,
    WOLFSSL_DH * dh
)
```

Computes shared secret from peer's public key.

Parameters:

- **key** Output buffer for shared secret
- **pub** Peer's public key
- **dh** DH structure with private key

See: wolfSSL_DH_generate_key

Return:

- Length of shared secret on success
- -1 on failure

Example

```
WOLFSSL_DH* dh = wolfSSL_DH_new();
wolfSSL_DH_generate_key(dh);
byte secret[256];
WOLFSSL_BIGNUM* peerPub = NULL; // peer's public key
int secretSz = wolfSSL_DH_compute_key(secret, peerPub, dh);
```

C.18.2.18 function wolfSSL_DH_compute_key_padded

```
int wolfSSL_DH_compute_key_padded(
    unsigned char * key,
    const WOLFSSL_BIGNUM * otherPub,
    WOLFSSL_DH * dh
)
```

Computes shared secret with zero-padding to DH size.

Parameters:

- **key** Output buffer for shared secret
- **otherPub** Peer's public key
- **dh** DH structure with private key

See: wolfSSL_DH_compute_key

Return:

- Length of shared secret on success
- -1 on failure

Example

```
WOLFSSL_DH* dh = wolfSSL_DH_new();
wolfSSL_DH_generate_key(dh);
byte secret[256];
WOLFSSL_BIGNUM* peerPub = NULL;
int secretSz = wolfSSL_DH_compute_key_padded(secret, peerPub, dh);
```

C.18.2.19 function wolfSSL_DH_LoadDer

```
int wolfSSL_DH_LoadDer(
    WOLFSSL_DH * dh,
    const unsigned char * derBuf,
    int derSz
)
```

Loads DH parameters from DER buffer.

Parameters:

- **dh** DH structure to load into
- **derBuf** DER-encoded DH parameters
- **derSz** Size of DER buffer

See: wolfSSL_DH_new

Return:

- WOLFSSL_SUCCESS on success
- WOLFSSL_FAILURE on failure

Example

```
WOLFSSL_DH* dh = wolfSSL_DH_new();
byte derBuf[256];
int ret = wolfSSL_DH_LoadDer(dh, derBuf, sizeof(derBuf));
```

C.18.2.20 function wolfSSL_DH_set_length

```
int wolfSSL_DH_set_length(
    WOLFSSL_DH * dh,
    long len
)
```

Sets optional private key length.

Parameters:

- **dh** DH structure
- **len** Private key length in bits

See: wolfSSL_DH_generate_key

Return:

- 1 on success
- 0 on failure

Example

```
WOLFSSL_DH* dh = wolfSSL_DH_new();
int ret = wolfSSL_DH_set_length(dh, 256);
```

C.18.2.21 function wolfSSL_DH_set0_pqg

```
int wolfSSL_DH_set0_pqg(
    WOLFSSL_DH * dh,
    WOLFSSL_BIGNUM * p,
    WOLFSSL_BIGNUM * q,
    WOLFSSL_BIGNUM * g
)
```

Sets DH parameters p, q, and g.

Parameters:

- **dh** DH structure
- **p** Prime modulus (takes ownership)
- **q** Subgroup order (takes ownership, can be NULL)
- **g** Generator (takes ownership)

See: wolfSSL_DH_generate_key

Return:

- 1 on success
- 0 on failure

Example

```
WOLFSSL_DH* dh = wolfSSL_DH_new();
WOLFSSL_BIGNUM *p = wolfSSL_BN_new();
WOLFSSL_BIGNUM *g = wolfSSL_BN_new();
// set p and g values
int ret = wolfSSL_DH_set0_pqg(dh, p, NULL, g);
```

C.18.2.22 function wolfSSL_DH_get_2048_256

```
WOLFSSL_DH * wolfSSL_DH_get_2048_256(
    void
)
```

Returns DH parameters for 2048-bit MODP group with 256-bit subgroup.

See: wolfSSL_DH_new_by_nid

Return:

- Pointer to WOLFSSL_DH on success
- NULL on failure

Example

```
WOLFSSL_DH* dh = wolfSSL_DH_get_2048_256();
if (dh == NULL) {
    // error getting standard group
}
```

C.18.2.23 function wc_Dh_ffdhe2048_Get

```
const DhParams * wc_Dh_ffdhe2048_Get(
    void
)
```

Returns FFDHE 2048-bit group parameters.

See:

- wc_Dh_ffdhe3072_Get
- wc_Dh_ffdhe4096_Get
- wc_Dh_ffdhe6144_Get
- wc_Dh_ffdhe8192_Get

Return:

- Pointer to DhParams structure
- NULL if not compiled with HAVE_FFDHE_2048

Example

```
const DhParams* params = wc_Dh_ffdhe2048_Get();
if (params != NULL) {
    // use params
}
```

C.18.2.24 function wc_Dh_ffdhe3072_Get

```
const DhParams * wc_Dh_ffdhe3072_Get(
    void
)
```

Returns FFDHE 3072-bit group parameters.

See:

- wc_Dh_ffdhe2048_Get
- wc_Dh_ffdhe4096_Get
- wc_Dh_ffdhe6144_Get
- wc_Dh_ffdhe8192_Get

Return:

- Pointer to DhParams structure
- NULL if not compiled with HAVE_FFDHE_3072

Example

```
const DhParams* params = wc_Dh_ffdhe3072_Get();
if (params != NULL) {
    // use params
}
```

C.18.2.25 function wc_Dh_ffdhe4096_Get

```
const DhParams * wc_Dh_ffdhe4096_Get(
    void
)
```

Returns FFDHE 4096-bit group parameters.

See:

- wc_Dh_ffdhe2048_Get
- wc_Dh_ffdhe3072_Get
- wc_Dh_ffdhe6144_Get
- wc_Dh_ffdhe8192_Get

Return:

- Pointer to DhParams structure
- NULL if not compiled with HAVE_FFDHE_4096

Example

```
const DhParams* params = wc_Dh_ffdhe4096_Get();
if (params != NULL) {
    // use params
}
```

C.18.2.26 function wc_Dh_ffdhe6144_Get

```
const DhParams * wc_Dh_ffdhe6144_Get(
    void
)
```

Returns FFDHE 6144-bit group parameters.

See:

- wc_Dh_ffdhe2048_Get
- wc_Dh_ffdhe3072_Get
- wc_Dh_ffdhe4096_Get
- wc_Dh_ffdhe8192_Get

Return:

- Pointer to DhParams structure
- NULL if not compiled with HAVE_FFDHE_6144

Example

```
const DhParams* params = wc_Dh_ffdhe6144_Get();
if (params != NULL) {
    // use params
}
```

C.18.2.27 function wc_Dh_ffdhe8192_Get

```
const DhParams * wc_Dh_ffdhe8192_Get(
    void
)
```

Returns FFDHE 8192-bit group parameters.

See:

- wc_Dh_ffdhe2048_Get
- wc_Dh_ffdhe3072_Get
- wc_Dh_ffdhe4096_Get
- wc_Dh_ffdhe6144_Get

Return:

- Pointer to DhParams structure
- NULL if not compiled with HAVE_FFDHE_8192

Example

```
const DhParams* params = wc_Dh_ffdhe8192_Get();
if (params != NULL) {
    // use params
}
```

C.18.2.28 function wc_InitDhKey_ex

```
int wc_InitDhKey_ex(
    DhKey * key,
    void * heap,
    int devId
)
```

Initializes DH key with heap hint and device ID.

Parameters:

- **key** DH key to initialize
- **heap** Heap hint for memory allocation
- **devId** Device ID for hardware acceleration

See:

- wc_InitDhKey
- wc_FreeDhKey

Return:

- 0 on success
- BAD_FUNC_ARG if key is NULL

Example

```
DhKey key;
int ret = wc_InitDhKey_ex(&key, NULL, INVALID_DEVID);
if (ret != 0) {
    // error initializing key
}
```

C.18.2.29 function wc_DhAgree_ct

```
int wc_DhAgree_ct(
    DhKey * key,
    byte * agree,
    word32 * agreeSz,
    const byte * priv,
    word32 privSz,
    const byte * otherPub,
    word32 pubSz
)
```

Computes shared secret with constant-time operations.

Parameters:

- **key** DH key with parameters
- **agree** Output buffer for shared secret
- **agreeSz** Input: buffer size, Output: secret size
- **priv** Private key
- **privSz** Private key size
- **otherPub** Peer's public key
- **pubSz** Peer's public key size

See: wc_DhAgree

Return:

- 0 on success
- BAD_FUNC_ARG if parameters are invalid
- BUFFER_E if output buffer too small

Example

```
DhKey key;
byte agree[256], priv[256], pub[256];
word32 agreeSz = sizeof(agree);
int ret = wc_DhAgree_ct(&key, agree, &agreeSz, priv,
                        sizeof(priv), pub, sizeof(pub));
```

C.18.2.30 function wc_DhSetNamedKey

```
int wc_DhSetNamedKey(
    DhKey * key,
```

```
    int name
)
```

Sets DH key to use named group parameters.

Parameters:

- **key** DH key to configure
- **name** Named group identifier

See: wc_DhGetNamedKeyParamSize

Return:

- 0 on success
- BAD_FUNC_ARG if parameters are invalid

Example

```
DhKey key;
wc_InitDhKey(&key);
int ret = wc_DhSetNamedKey(&key, WC_FFDHE_2048);
```

C.18.2.31 function wc_DhGetNamedKeyParamSize

```
int wc_DhGetNamedKeyParamSize(
    int name,
    word32 * p,
    word32 * g,
    word32 * q
)
```

Gets parameter sizes for named group.

Parameters:

- **name** Named group identifier
- **p** Output for prime size
- **g** Output for generator size
- **q** Output for subgroup order size

See: wc_DhSetNamedKey

Return:

- 0 on success
- BAD_FUNC_ARG if parameters are invalid

Example

```
word32 pSz, gSz, qSz;
int ret = wc_DhGetNamedKeyParamSize(WC_FFDHE_2048, &pSz, &gSz,
                                    &qSz);
```

C.18.2.32 function wc_DhGetNamedKeyMinSize

```
word32 wc_DhGetNamedKeyMinSize(
    int name
)
```

Gets minimum key size for named group.

Parameters:

- **name** Named group identifier

See: wc_DhSetNamedKey

Return:

- Minimum key size in bits
- 0 if invalid name

Example

```
word32 minSize = wc_DhGetNamedKeyMinSize(WC_FFDHE_2048);
```

C.18.2.33 function wc_DhCmpNamedKey

```
int wc_DhCmpNamedKey(
    int name,
    int noQ,
    const byte * p,
    word32 pSz,
    const byte * g,
    word32 gSz,
    const byte * q,
    word32 qSz
)
```

Compares parameters against named group.

Parameters:

- **name** Named group identifier
- **noQ** 1 to skip q comparison
- **p** Prime modulus
- **pSz** Prime size
- **g** Generator
- **gSz** Generator size
- **q** Subgroup order
- **qSz** Subgroup order size

See: wc_DhSetNamedKey

Return:

- 0 if parameters match named group
- Non-zero if parameters don't match

Example

```
byte p[256], g[256];
int ret = wc_DhCmpNamedKey(WC_FFDHE_2048, 1, p, sizeof(p),
                           g, sizeof(g), NULL, 0);
```

C.18.2.34 function wc_DhCopyNamedKey

```
int wc_DhCopyNamedKey(
    int name,
    byte * p,
    word32 * pSz,
    byte * g,
    word32 * gSz,
    byte * q,
```

```
    word32 * qSz
)
```

Copies named group parameters to buffers.

Parameters:

- **name** Named group identifier
- **p** Output buffer for prime
- **pSz** Input: buffer size, Output: prime size
- **g** Output buffer for generator
- **gSz** Input: buffer size, Output: generator size
- **q** Output buffer for subgroup order
- **qSz** Input: buffer size, Output: subgroup order size

See: wc_DhSetNamedKey

Return:

- 0 on success
- BAD_FUNC_ARG if parameters are invalid
- BUFFER_E if buffers too small

Example

```
byte p[512], g[512], q[512];
word32 pSz = sizeof(p), gSz = sizeof(g), qSz = sizeof(q);
int ret = wc_DhCopyNamedKey(WC_FFDHE_2048, p, &pSz, g, &gSz,
                            q, &qSz);
```

C.18.2.35 function wc_DhGeneratePublic

```
int wc_DhGeneratePublic(
    DhKey * key,
    byte * priv,
    word32 privSz,
    byte * pub,
    word32 * pubSz
)
```

Generates public key from private key.

Parameters:

- **key** DH key with parameters set
- **priv** Private key
- **privSz** Private key size
- **pub** Output buffer for public key
- **pubSz** Input: buffer size, Output: public key size

See: wc_DhGenerateKeyPair

Return:

- 0 on success
- BAD_FUNC_ARG if parameters are invalid

Example

```
DhKey key;
byte priv[256], pub[256];
word32 pubSz = sizeof(pub);
```

```
int ret = wc_DhGeneratePublic(&key, priv, sizeof(priv), pub,
                              &pubSz);
```

C.18.2.36 function wc_DhImportKeyPair

```
int wc_DhImportKeyPair(
    DhKey * key,
    const byte * priv,
    word32 privSz,
    const byte * pub,
    word32 pubSz
)
```

Imports private and/or public key into DH key.

Parameters:

- **key** DH key to import into
- **priv** Private key (can be NULL)
- **privSz** Private key size
- **pub** Public key (can be NULL)
- **pubSz** Public key size

See: wc_DhExportKeyPair

Return:

- 0 on success
- BAD_FUNC_ARG if parameters are invalid

Example

```
DhKey key;
byte priv[256], pub[256];
int ret = wc_DhImportKeyPair(&key, priv, sizeof(priv), pub,
                             sizeof(pub));
```

C.18.2.37 function wc_DhExportKeyPair

```
int wc_DhExportKeyPair(
    DhKey * key,
    byte * priv,
    word32 * pPrivSz,
    byte * pub,
    word32 * pPubSz
)
```

Exports private and public key from DH key.

Parameters:

- **key** DH key to export from
- **priv** Output buffer for private key
- **pPrivSz** Input: buffer size, Output: private key size
- **pub** Output buffer for public key
- **pPubSz** Input: buffer size, Output: public key size

See: wc_DhImportKeyPair

Return:

- 0 on success
- BAD_FUNC_ARG if parameters are invalid
- BUFFER_E if buffers too small

Example

```
DhKey key;
byte priv[256], pub[256];
word32 privSz = sizeof(priv), pubSz = sizeof(pub);
int ret = wc_DhExportKeyPair(&key, priv, &privSz, pub, &pubSz);
```

C.18.2.38 function wc_DhCheckPubValue

```
int wc_DhCheckPubValue(
    const byte * prime,
    word32 primeSz,
    const byte * pub,
    word32 pubSz
)
```

Validates public key value.

Parameters:

- **prime** Prime modulus
- **primeSz** Prime size
- **pub** Public key to validate
- **pubSz** Public key size

See: wc_DhCheckPubKey

Return:

- 0 if public key is valid
- BAD_FUNC_ARG if parameters are invalid
- MP_VAL if public key is invalid

Example

```
byte prime[256], pub[256];
int ret = wc_DhCheckPubValue(prime, sizeof(prime), pub,
                             sizeof(pub));
if (ret != 0) {
    // invalid public key
}
```

C.18.2.39 function wc_DhCheckKeyPair

```
int wc_DhCheckKeyPair(
    DhKey * key,
    const byte * pub,
    word32 pubSz,
    const byte * priv,
    word32 privSz
)
```

Checks DH keys for pair-wise consistency per process in SP 800-56Ar3, section 5.6.2.1.4, method (b) for FFC.

C.18.2.40 function wc_DhCheckPrivKey

```
int wc_DhCheckPrivKey(
    DhKey * key,
    const byte * priv,
    word32 pubSz
)
```

Check DH private key for invalid numbers.

C.18.2.41 function wc_DhCheckPrivKey_ex

```
int wc_DhCheckPrivKey_ex(
    DhKey * key,
    const byte * priv,
    word32 pubSz,
    const byte * prime,
    word32 primeSz
)
```

C.18.2.42 function wc_DhCheckPubKey

```
int wc_DhCheckPubKey(
    DhKey * key,
    const byte * pub,
    word32 pubSz
)
```

C.18.2.43 function wc_DhCheckPubKey_ex

```
int wc_DhCheckPubKey_ex(
    DhKey * key,
    const byte * pub,
    word32 pubSz,
    const byte * prime,
    word32 primeSz
)
```

C.18.2.44 function wc_DhExportParamsRaw

```
int wc_DhExportParamsRaw(
    DhKey * dh,
    byte * p,
    word32 * pSz,
    byte * q,
    word32 * qSz,
    byte * g,
    word32 * gSz
)
```

C.18.2.45 function wc_DhGenerateParams

```
int wc_DhGenerateParams(
    WC_RNG * rng,
    int modSz,
```

```
    DhKey * dh
)
```

C.18.2.46 function wc_DhSetCheckKey

```
int wc_DhSetCheckKey(
    DhKey * key,
    const byte * p,
    word32 pSz,
    const byte * g,
    word32 gSz,
    const byte * q,
    word32 qSz,
    int trusted,
    WC_RNG * rng
)
```

C.18.2.47 function wc_DhSetKey_ex

```
int wc_DhSetKey_ex(
    DhKey * key,
    const byte * p,
    word32 pSz,
    const byte * g,
    word32 gSz,
    const byte * q,
    word32 qSz
)
```

Note: Parameters that are not one of the built-in FFDHE groups are untrusted, so the modulus is tested for primality with random Miller-Rabin witnesses. This function takes no RNG, so it instantiates a temporary one, which allocates memory and consumes entropy on every such call. When neither is available the test falls back to fixed witness bases, which a purpose-built composite can defeat. Call wc_DhSetCheckKey() with an initialized WC_RNG to supply your own.

C.18.3 Source code

```
int wc_InitDhKey(DhKey* key);

int wc_FreeDhKey(DhKey* key);

int wc_DhGenerateKeyPair(DhKey* key, WC_RNG* rng, byte* priv,
                                 word32* privSz, byte* pub, word32* pubSz);

int wc_DhAgree(DhKey* key, byte* agree, word32* agreeSz,
                       const byte* priv, word32 privSz, const byte* otherPub,
                       word32 pubSz);

int wc_DhKeyDecode(const byte* input, word32* inOutIdx, DhKey* key,
                           word32 inSz);

int wc_DhSetKey(DhKey* key, const byte* p, word32 pSz, const byte* g,
                        word32 gSz);
```

```
int wc_DhParamsLoad(const byte* input, word32 inSz, byte* p,
                            word32* pInOutSz, byte* g, word32* gInOutSz);

int wolfSSL_i2d_DHparams(const WOLFSSL_DH *dh, unsigned char **out);

WOLFSSL_DH* wolfSSL_DH_new(void);

WOLFSSL_DH* wolfSSL_DH_new_by_nid(int nid);

void wolfSSL_DH_free(WOLFSSL_DH* dh);

WOLFSSL_DH* wolfSSL_DH_dup(WOLFSSL_DH* dh);

int wolfSSL_DH_up_ref(WOLFSSL_DH* dh);

int wolfSSL_DH_check(const WOLFSSL_DH *dh, int *codes);

int wolfSSL_DH_size(WOLFSSL_DH* dh);

int wolfSSL_DH_generate_key(WOLFSSL_DH* dh);

int wolfSSL_DH_compute_key(unsigned char* key,
                           const WOLFSSL_BIGNUM* pub, WOLFSSL_DH* dh);

int wolfSSL_DH_compute_key_padded(unsigned char* key,
                                  const WOLFSSL_BIGNUM* otherPub,
                                  WOLFSSL_DH* dh);

int wolfSSL_DH_LoadDer(WOLFSSL_DH* dh, const unsigned char* derBuf,
                       int derSz);

int wolfSSL_DH_set_length(WOLFSSL_DH* dh, long len);

int wolfSSL_DH_set0_pqg(WOLFSSL_DH *dh, WOLFSSL_BIGNUM *p,
                        WOLFSSL_BIGNUM *q, WOLFSSL_BIGNUM *g);

WOLFSSL_DH* wolfSSL_DH_get_2048_256(void);

const DhParams* wc_Dh_ffdhe2048_Get(void);

const DhParams* wc_Dh_ffdhe3072_Get(void);

const DhParams* wc_Dh_ffdhe4096_Get(void);

const DhParams* wc_Dh_ffdhe6144_Get(void);

const DhParams* wc_Dh_ffdhe8192_Get(void);

int wc_InitDhKey_ex(DhKey* key, void* heap, int devId);

int wc_DhAgree_ct(DhKey* key, byte* agree, word32 *agreeSz,
                  const byte* priv, word32 privSz,
                  const byte* otherPub, word32 pubSz);
```

```
int wc_DhSetNamedKey(DhKey* key, int name);

int wc_DhGetNamedKeyParamSize(int name, word32* p, word32* g,
                              word32* q);

word32 wc_DhGetNamedKeyMinSize(int name);

int wc_DhCmpNamedKey(int name, int noQ, const byte* p, word32 pSz,
                     const byte* g, word32 gSz, const byte* q,
                     word32 qSz);

int wc_DhCopyNamedKey(int name, byte* p, word32* pSz, byte* g,
                      word32* gSz, byte* q, word32* qSz);

int wc_DhGeneratePublic(DhKey* key, byte* priv, word32 privSz,
                        byte* pub, word32* pubSz);

int wc_DhImportKeyPair(DhKey* key, const byte* priv, word32 privSz,
                       const byte* pub, word32 pubSz);

int wc_DhExportKeyPair(DhKey* key, byte* priv, word32* pPrivSz,
                       byte* pub, word32* pPubSz);

int wc_DhCheckPubValue(const byte* prime, word32 primeSz,
                       const byte* pub, word32 pubSz);

int wc_DhCheckKeyPair(DhKey* key, const byte* pub, word32 pubSz,
                      const byte* priv, word32 privSz);

int wc_DhCheckPrivKey(DhKey* key, const byte* priv, word32 pubSz);

int wc_DhCheckPrivKey_ex(DhKey* key, const byte* priv, word32 pubSz,
                         const byte* prime, word32 primeSz);

int wc_DhCheckPubKey(DhKey* key, const byte* pub, word32 pubSz);

int wc_DhCheckPubKey_ex(DhKey* key, const byte* pub, word32 pubSz,
                        const byte* prime, word32 primeSz);

int wc_DhExportParamsRaw(DhKey* dh, byte* p, word32* pSz,
                         byte* q, word32* qSz, byte* g, word32* gSz);

int wc_DhGenerateParams(WC_RNG *rng, int modSz, DhKey *dh);

int wc_DhSetCheckKey(DhKey* key, const byte* p, word32 pSz,
                     const byte* g, word32 gSz, const byte* q, word32 qSz,
                     int trusted, WC_RNG* rng);

int wc_DhSetKey_ex(DhKey* key, const byte* p, word32 pSz,
                   const byte* g, word32 gSz, const byte* q, word32 qSz);

int wc_FreeDhKey(DhKey* key);
```

C.19 doxygen_groups.h

C.20 doxygen_pages.h

C.21 dsa.h

C.21.1 Functions

	Name
int	**wc_InitDsaKey**(DsaKey * key)This function initializes a DsaKey object in order to use it for authentication via the Digital Signature Algorithm (DSA).
void	**wc_FreeDsaKey**(DsaKey * key)This function frees a DsaKey object after it has been used.
int	**wc_DsaSign**(const byte * digest, byte * out, DsaKey * key, WC_RNG * rng)This function signs the input digest and stores the result in the output buffer, out.
int	**wc_DsaVerify**(const byte * digest, const byte * sig, DsaKey * key, int * answer)This function verifies the signature of a digest, given a private key. It stores whether the key properly verifies in the answer parameter, with 1 corresponding to a successful verification, and 0 corresponding to failed verification.
int	**wc_DsaCheckPubKey**(DsaKey * key)This function validates DSA domain parameters and the public key contained in the given DsaKey. It performs the following checks (a subset of the requirements in FIPS 186_4 and NIST SP 800_89): p > 1 and q > 1; q divides (p _ 1); 1 < g < p; 1 < y < p; g^q mod p == 1 (so the multiplicative order of g divides q); and y^q mod p == 1 (so the multiplicative order of y divides q). The verify routines (wc_DsaVerify, wc_DsaVerify_ex) call this internally before any signature math runs, but the function is also exposed so callers can validate keys at import or decode time.
int	**wc_DsaPublicKeyDecode**(const byte * input, word32 * inOutIdx, DsaKey * key, word32 inSz)This function decodes a DER formatted certificate buffer containing a DSA public key, and stores the key in the given DsaKey structure. It also sets the inOutIdx parameter according to the length of the input read.
int	**wc_DsaPrivateKeyDecode**(const byte * input, word32 * inOutIdx, DsaKey * key, word32 inSz)This function decodes a DER formatted certificate buffer containing a DSA private key, and stores the key in the given DsaKey structure. It also sets the inOutIdx parameter according to the length of the input read.

	Name
int	**wc_DsaKeyToDer**(DsaKey * key, byte * output, word32 outLen)Convert DsaKey key to DER format, write to output (outLen), return bytes written.
int	**wc_MakeDsaKey**(WC_RNG * rng, DsaKey * dsa)Create a DSA key.
int	**wc_MakeDsaParameters**(WC_RNG * rng, int modulus_size, DsaKey * dsa)FIPS 186_4 defines valid for modulus_size values as (1024, 160) (2048, 256) (3072, 256)
int	**wc_InitDsaKey_h**(DsaKey * key, void * h)Initializes DSA key with heap hint.
int	**wc_DsaSign_ex**(const byte * digest, word32 digestSz, byte * out, DsaKey * key, WC_RNG * rng)Signs digest with extended parameters.
int	**wc_DsaVerify_ex**(const byte * digest, word32 digestSz, const byte * sig, DsaKey * key, int * answer)Verifies signature with extended parameters.
int	**wc_SetDsaPublicKey**(byte * output, DsaKey * key, int outLen, int with_header)Sets DSA public key in output buffer.
int	**wc_DsaKeyToPublicDer**(DsaKey * key, byte * output, word32 outLen)Converts DSA key to public DER format.
int	**wc_DsaImportParamsRaw**(DsaKey * dsa, const char * p, const char * q, const char * g)Imports DSA parameters from raw format. The parameters p, q, and g must be provided as ASCII hexadecimal strings (without 0x prefix). These represent the DSA domain parameters: p is the prime modulus, q is the prime divisor (subgroup order), and g is the generator.
int	**wc_DsaImportParamsRawCheck**(DsaKey * dsa, const char * p, const char * q, const char * g, int trusted, WC_RNG * rng)Imports DSA parameters from raw format with optional validation. The parameters p, q, and g must be provided as ASCII hexadecimal strings (without 0x prefix). The trusted parameter controls whether the prime p is validated: when trusted=1, prime checking is skipped (use when parameters come from a trusted source); when trusted=0, performs full primality testing on p (recommended for untrusted sources).
int	**wc_DsaExportParamsRaw**(DsaKey * dsa, byte * p, word32 * pSz, byte * q, word32 * qSz, byte * g, word32 * gSz)Exports DSA parameters to raw format.
int	**wc_DsaExportKeyRaw**(DsaKey * dsa, byte * x, word32 * xSz, byte * y, word32 * ySz)Exports DSA key to raw format.

COPYRIGHT ©2024 wolfSSL Inc.

C.21.2 Functions Documentation

C.21.2.1 function wc_InitDsaKey

```
int wc_InitDsaKey(
    DsaKey * key
)
```

This function initializes a DsaKey object in order to use it for authentication via the Digital Signature Algorithm (DSA).

Parameters:

- **key** pointer to the DsaKey structure to initialize

See: wc_FreeDsaKey

Return:

- 0 Returned on success.
- BAD_FUNC_ARG Returned if a NULL key is passed in.

Example

```
DsaKey key;
int ret;
ret = wc_InitDsaKey(&key); // initialize DSA key
```

C.21.2.2 function wc_FreeDsaKey

```
void wc_FreeDsaKey(
    DsaKey * key
)
```

This function frees a DsaKey object after it has been used.

Parameters:

- **key** pointer to the DsaKey structure to free

See: wc_FreeDsaKey

Return: none No returns.

Example

```
DsaKey key;
// initialize key, use for authentication
...
wc_FreeDsaKey(&key); // free DSA key
```

C.21.2.3 function wc_DsaSign

```
int wc_DsaSign(
    const byte * digest,
    byte * out,
    DsaKey * key,
    WC_RNG * rng
)
```

This function signs the input digest and stores the result in the output buffer, out.

Parameters:

- **digest** pointer to the hash to sign
- **out** pointer to the buffer in which to store the signature
- **key** pointer to the initialized DsaKey structure with which to generate the signature
- **rng** pointer to an initialized RNG to use with the signature generation

See: wc_DsaVerify

Return:

- 0 Returned on successfully signing the input digest
- MP_INIT_E may be returned if there is an error in processing the DSA signature.
- MP_READ_E may be returned if there is an error in processing the DSA signature.
- MP_CMP_E may be returned if there is an error in processing the DSA signature.
- MP_INVMOD_E may be returned if there is an error in processing the DSA signature.
- MP_EXPTMOD_E may be returned if there is an error in processing the DSA signature.
- MP_MOD_E may be returned if there is an error in processing the DSA signature.
- MP_MUL_E may be returned if there is an error in processing the DSA signature.
- MP_ADD_E may be returned if there is an error in processing the DSA signature.
- MP_MULMOD_E may be returned if there is an error in processing the DSA signature.
- MP_TO_E may be returned if there is an error in processing the DSA signature.
- MP_MEM may be returned if there is an error in processing the DSA signature.

Example

```
DsaKey key;
// initialize DSA key, load private Key
int ret;
WC_RNG rng;
wc_InitRng(&rng);
byte hash[] = { // initialize with hash digest };
byte signature[40]; // signature will be 40 bytes (320 bits)

ret = wc_DsaSign(hash, signature, &key, &rng);
if (ret != 0) {
    // error generating DSA signature
}
```

C.21.2.4 function wc_DsaVerify

```
int wc_DsaVerify(
    const byte * digest,
    const byte * sig,
    DsaKey * key,
    int * answer
)
```

This function verifies the signature of a digest, given a private key. It stores whether the key properly verifies in the answer parameter, with 1 corresponding to a successful verification, and 0 corresponding to failed verification.

Parameters:

- **digest** pointer to the digest containing the subject of the signature
- **sig** pointer to the buffer containing the signature to verify
- **key** pointer to the initialized DsaKey structure with which to verify the signature
- **answer** pointer to an integer which will store whether the verification was successful

See: wc_DsaSign

Return:

- 0 Returned on successfully processing the verify request. Note: this does not mean that the signature is verified, only that the function succeeded
- MP_INIT_E may be returned if there is an error in processing the DSA signature.
- MP_READ_E may be returned if there is an error in processing the DSA signature.
- MP_CMP_E may be returned if there is an error in processing the DSA signature.
- MP_INVMOD_E may be returned if there is an error in processing the DSA signature.
- MP_EXPTMOD_E may be returned if there is an error in processing the DSA signature.
- MP_MOD_E may be returned if there is an error in processing the DSA signature.
- MP_MUL_E may be returned if there is an error in processing the DSA signature.
- MP_ADD_E may be returned if there is an error in processing the DSA signature.
- MP_MULMOD_E may be returned if there is an error in processing the DSA signature.
- MP_TO_E may be returned if there is an error in processing the DSA signature.
- MP_MEM may be returned if there is an error in processing the DSA signature.

Example

```
DsaKey key;
// initialize DSA key, load public Key

int ret;
int verified;
byte hash[] = { // initialize with hash digest };
byte signature[] = { // initialize with signature to verify };
ret = wc_DsaVerify(hash, signature, &key, &verified);
if (ret != 0) {
    // error processing verify request
} else if (answer == 0) {
    // invalid signature
}
```

C.21.2.5 function wc_DsaCheckPubKey

```
int wc_DsaCheckPubKey(
    DsaKey * key
)
```

This function validates DSA domain parameters and the public key contained in the given DsaKey. It performs the following checks (a subset of the requirements in FIPS 186-4 and NIST SP 800-89): p > 1 and q > 1; q divides (p - 1); 1 < g < p; 1 < y < p; g^q mod p == 1 (so the multiplicative order of g divides q); and y^q mod p == 1 (so the multiplicative order of y divides q). The verify routines (wc_DsaVerify, wc_DsaVerify_ex) call this internally before any signature math runs, but the function is also exposed so callers can validate keys at import or decode time.

Parameters:

- **key** pointer to a DsaKey structure populated with p, q, g, and y

See:

- wc_DsaVerify
- wc_DsaPublicKeyDecode
- wc_DsaImportParamsRaw
- wc_DsaImportParamsRawCheck

Return:

- 0 Returned when the key and domain parameters pass validation.

- BAD_FUNC_ARG Returned when key is NULL, or when the key fails any of the validation checks listed above.
- MEMORY_E Returned on memory allocation failure (small-stack builds only).
- MP_INIT_E Returned when mp_int initialization fails.
- Other negative MP error codes (e.g. MP_EXPTMOD_E) Returned on internal big-integer failures.

Note: this function does not run primality tests on p or q. Full FIPS 186-4 domain-parameter validation additionally requires that p and q be prime; callers that need that level of assurance should use wc_DsaImportParamsRawCheck() (which validates p) and/or run mp_prime_is_prime_ex() on q at import time.

Example

```
DsaKey key;
int ret;
wc_InitDsaKey(&key);
// ... import or decode the public key into &key ...
ret = wc_DsaCheckPubKey(&key);
if (ret != 0) {
    // domain parameters or public key are invalid; reject
}
wc_FreeDsaKey(&key);
```

C.21.2.6 function wc_DsaPublicKeyDecode

```
int wc_DsaPublicKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    DsaKey * key,
    word32 inSz
)
```

This function decodes a DER formatted certificate buffer containing a DSA public key, and stores the key in the given DsaKey structure. It also sets the inOutIdx parameter according to the length of the input read.

Parameters:

- **input** pointer to the buffer containing the DER formatted DSA public key
- **inOutIdx** pointer to an integer in which to store the final index of the certificate read
- **key** pointer to the DsaKey structure in which to store the public key
- **inSz** size of the input buffer

See:

- wc_InitDsaKey
- wc_DsaPrivateKeyDecode

Return:

- 0 Returned on successfully setting the public key for the DsaKey object
- ASN_PARSE_E Returned if there is an error in the encoding while reading the certificate buffer
- ASN_DH_KEY_E Returned if one of the DSA parameters is incorrectly formatted

Example

```
int ret, idx=0;

DsaKey key;
wc_InitDsaKey(&key);
```

```
byte derBuff[] = { // DSA public key};
ret = wc_DsaPublicKeyDecode(derBuff, &idx, &key, inSz);
if (ret != 0) {
    // error reading public key
}
```

C.21.2.7 function wc_DsaPrivateKeyDecode

```
int wc_DsaPrivateKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    DsaKey * key,
    word32 inSz
)
```

This function decodes a DER formatted certificate buffer containing a DSA private key, and stores the key in the given DsaKey structure. It also sets the inOutIdx parameter according to the length of the input read.

Parameters:

- **input** pointer to the buffer containing the DER formatted DSA private key
- **inOutIdx** pointer to an integer in which to store the final index of the certificate read
- **key** pointer to the DsaKey structure in which to store the private key
- **inSz** size of the input buffer

See:

- wc_InitDsaKey
- wc_DsaPublicKeyDecode

Return:

- 0 Returned on successfully setting the private key for the DsaKey object
- ASN_PARSE_E Returned if there is an error in the encoding while reading the certificate buffer
- ASN_DH_KEY_E Returned if one of the DSA parameters is incorrectly formatted

Example

```
int ret, idx=0;

DsaKey key;
wc_InitDsaKey(&key);
byte derBuff[] = { // DSA private key };
ret = wc_DsaPrivateKeyDecode(derBuff, &idx, &key, inSz);
if (ret != 0) {
    // error reading private key
}
```

C.21.2.8 function wc_DsaKeyToDer

```
int wc_DsaKeyToDer(
    DsaKey * key,
    byte * output,
    word32 outLen
)
```

Convert DsaKey key to DER format, write to output (outLen), return bytes written.

Parameters:

- **key** Pointer to DsaKey structure to convert.
- **output** Pointer to output buffer for converted key.
- **outLen** Length of output buffer.

See:

- wc_InitDsaKey
- wc_FreeDsaKey
- wc_MakeDsaKey

Return:

- outLen Success, number of bytes written
- BAD_FUNC_ARG key or output are null or key->type is not DSA_PRIVATE.
- MEMORY_E Error allocating memory.

Example

```
DsaKey key;
WC_RNG rng;
int derSz;
int bufferSize = // Sufficient buffer size;
byte der[bufferSize];

wc_InitDsaKey(&key);
wc_InitRng(&rng);
wc_MakeDsaKey(&rng, &key);
derSz = wc_DsaKeyToDer(&key, der, bufferSize);
```

C.21.2.9 function wc_MakeDsaKey

```
int wc_MakeDsaKey(
    WC_RNG * rng,
    DsaKey * dsa
)
```

Create a DSA key.

Parameters:

- **rng** Pointer to WC_RNG structure.
- **dsa** Pointer to DsaKey structure.

See:

- wc_InitDsaKey
- wc_FreeDsaKey
- wc_DsaSign

Return:

- MP_OKAY Success
- BAD_FUNC_ARG Either rng or dsa is null.
- MEMORY_E Couldn't allocate memory for buffer.
- MP_INIT_E Error initializing mp_int

Example

```
WC_RNG rng;
DsaKey dsa;
wc_InitRng(&rng);
wc_InitDsa(&dsa);
```

```
if(wc_MakeDsaKey(&rng, &dsa) != 0)
{
    // Error creating key
}
```

C.21.2.10 function wc_MakeDsaParameters

```
int wc_MakeDsaParameters(
    WC_RNG * rng,
    int modulus_size,
    DsaKey * dsa
)
```

FIPS 186-4 defines valid for modulus_size values as (1024, 160) (2048, 256) (3072, 256)

Parameters:

- **rng** pointer to wolfCrypt rng.
- **modulus_size** 1024, 2048, or 3072 are valid values.
- **dsa** Pointer to a DsaKey structure.

See:

- wc_MakeDsaKey
- wc_DsaKeyToDer
- wc_InitDsaKey

Return:

- 0 Success
- BAD_FUNC_ARG rng or dsa is null or modulus_size is invalid.
- MEMORY_E Error attempting to allocate memory.

Example

```
DsaKey key;
WC_RNG rng;
wc_InitDsaKey(&key);
wc_InitRng(&rng);
if(wc_MakeDsaParameters(&rng, 1024, &genKey) != 0)
{
    // Handle error
}
```

C.21.2.11 function wc_InitDsaKey_h

```
int wc_InitDsaKey_h(
    DsaKey * key,
    void * h
)
```

Initializes DSA key with heap hint.

Parameters:

- **key** DSA key structure
- **h** Heap hint for memory allocation

See: wc_InitDsaKey

Return:

- 0 on success
- negative on failure

Example

```
DsaKey key;
int ret = wc_InitDsaKey_h(&key, NULL);
```

C.21.2.12 function wc_DsaSign_ex

```
int wc_DsaSign_ex(
    const byte * digest,
    word32 digestSz,
    byte * out,
    DsaKey * key,
    WC_RNG * rng
)
```

Signs digest with extended parameters.

Parameters:

- **digest** Digest to sign
- **digestSz** Digest size
- **out** Output signature buffer
- **key** DSA key
- **rng** Random number generator

See: wc_DsaSign

Return:

- 0 on success
- negative on failure

Example

```
byte digest[WC_SHA_DIGEST_SIZE];
byte sig[40];
WC_RNG rng;
int ret = wc_DsaSign_ex(digest, sizeof(digest), sig, &key,
                        &rng);
```

C.21.2.13 function wc_DsaVerify_ex

```
int wc_DsaVerify_ex(
    const byte * digest,
    word32 digestSz,
    const byte * sig,
    DsaKey * key,
    int * answer
)
```

Verifies signature with extended parameters.

Parameters:

- **digest** Digest
- **digestSz** Digest size
- **sig** Signature buffer
- **key** DSA key

- **answer** Verification result

See: wc_DsaVerify

Return:

- 0 on success
- negative on failure

Example

```
byte digest[WC_SHA_DIGEST_SIZE];
byte sig[40];
int answer;
int ret = wc_DsaVerify_ex(digest, sizeof(digest), sig, &key,
                          &answer);
```

C.21.2.14 function wc_SetDsaPublicKey

```
int wc_SetDsaPublicKey(
    byte * output,
    DsaKey * key,
    int outLen,
    int with_header
)
```

Sets DSA public key in output buffer.

Parameters:

- **output** Output buffer
- **key** DSA key
- **outLen** Output buffer length
- **with_header** Include header flag

See: wc_DsaKeyToPublicDer

Return:

- Size on success
- negative on failure

Example

```
byte output[256];
int ret = wc_SetDsaPublicKey(output, &key, sizeof(output), 1);
```

C.21.2.15 function wc_DsaKeyToPublicDer

```
int wc_DsaKeyToPublicDer(
    DsaKey * key,
    byte * output,
    word32 outLen
)
```

Converts DSA key to public DER format.

Parameters:

- **key** DSA key
- **output** Output buffer
- **outLen** Output buffer length

See: wc_SetDsaPublicKey

Return:

- Size on success
- negative on failure

Example

```
DsaKey key;
WC_RNG rng;
byte output[256];

// Initialize key and RNG
wc_InitDsaKey(&key);
wc_InitRng(&rng);

// Generate DSA key or import existing key
wc_MakeDsaKey(&rng, &key);

// Convert to public DER format
int ret = wc_DsaKeyToPublicDer(&key, output, sizeof(output));
if (ret > 0) {
    // output contains DER encoded public key of size ret
}

wc_FreeDsaKey(&key);
wc_FreeRng(&rng);
```

C.21.2.16 function wc_DsaImportParamsRaw

```
int wc_DsaImportParamsRaw(
    DsaKey * dsa,
    const char * p,
    const char * q,
    const char * g
)
```

Imports DSA parameters from raw format. The parameters p, q, and g must be provided as ASCII hexadecimal strings (without 0x prefix). These represent the DSA domain parameters: p is the prime modulus, q is the prime divisor (subgroup order), and g is the generator.

Parameters:

- **dsa** DSA key structure (must be initialized)
- **p** P parameter as ASCII hex string (prime modulus)
- **q** Q parameter as ASCII hex string (prime divisor/subgroup order)
- **g** G parameter as ASCII hex string (generator)

See:

- wc_DsaImportParamsRawCheck
- wc_InitDsaKey

Return:

- 0 on success
- negative on failure

Example

```
DsaKey dsa;
wc_InitDsaKey(&dsa);

// DSA parameters as ASCII hexadecimal strings (example values)
const char* pStr = "E0A67598CD1B763BC98C8ABB333E5DDA0CD3AA0E5E1F"
                   "B5BA8A7B4EABC10BA338FAE06DD4B90FDA70D7CF0CB0"
                   "C638BE3341BEC0AF8A7330A3307DED2299A0EE606DF0"
                   "35177A239C34A912C202AA5F83B9C4A7CF0235B5316B"
                   "FC6EFB9A248411258B30B839AF172440F32563056CB6"
                   "7A861158DDD90E6A894C72A5BBEF9E286C6B";
const char* qStr = "E950511EAB424B9A19A2AEB4E159B7844C589C4F";
const char* gStr = "D29D5121B0423C2769AB21843E5A3240FF19CACC792D"
                   "C6E7925E6D1A4E6E4E3D119A3D133C8D3C8C8C8C8C8C"
                   "8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C"
                   "8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C";

int ret = wc_DsaImportParamsRaw(&dsa, pStr, qStr, gStr);
if (ret == 0) {
    // DSA parameters successfully imported
    // Can now use dsa for key generation or signing
}
wc_FreeDsaKey(&dsa);
```

C.21.2.17 function wc_DsaImportParamsRawCheck

```
int wc_DsaImportParamsRawCheck(
    DsaKey * dsa,
    const char * p,
    const char * q,
    const char * g,
    int trusted,
    WC_RNG * rng
)
```

Imports DSA parameters from raw format with optional validation. The parameters p, q, and g must be provided as ASCII hexadecimal strings (without 0x prefix). The trusted parameter controls whether the prime p is validated: when trusted=1, prime checking is skipped (use when parameters come from a trusted source); when trusted=0, performs full primality testing on p (recommended for untrusted sources).

Parameters:

- **dsa** DSA key structure (must be initialized)
- **p** P parameter as ASCII hex string (prime modulus)
- **q** Q parameter as ASCII hex string (prime divisor/subgroup order)
- **g** G parameter as ASCII hex string (generator)
- **trusted** If 1, skip prime validation (trusted source); if 0, perform full primality test on p
- **rng** Random number generator (required when trusted=0 for primality testing)

See:

- wc_DsaImportParamsRaw
- wc_InitDsaKey

Return:

- 0 on success
- DH_CHECK_PUB_E if p fails primality test (when trusted=0)

- negative on other failures

Example

```
DsaKey dsa;
WC_RNG rng;

// Initialize DSA key and RNG
wc_InitDsaKey(&dsa);
wc_InitRng(&rng);

// DSA parameters as ASCII hexadecimal strings
const char* pStr = "E0A67598CD1B763BC98C8ABB333E5DDA0CD3AA0E5E1F"
                   "B5BA8A7B4EABC10BA338FAE06DD4B90FDA70D7CF0CB0"
                   "C638BE3341BEC0AF8A7330A3307DED2299A0EE606DF0"
                   "35177A239C34A912C202AA5F83B9C4A7CF0235B5316B"
                   "FC6EFB9A248411258B30B839AF172440F32563056CB6"
                   "7A861158DDD90E6A894C72A5BBEF9E286C6B";
const char* qStr = "E950511EAB424B9A19A2AEB4E159B7844C589C4F";
const char* gStr = "D29D5121B0423C2769AB21843E5A3240FF19CACC792D"
                   "C6E7925E6D1A4E6E4E3D119A3D133C8D3C8C8C8C8C8C"
                   "8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C"
                   "8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C8C";

// Import with validation (trusted=0 performs primality test on p)
int ret = wc_DsaImportParamsRawCheck(&dsa, pStr, qStr, gStr, 0,
                                     &rng);
if (ret == 0) {
    // Parameters imported and validated successfully
}

wc_FreeDsaKey(&dsa);
wc_FreeRng(&rng);
```

C.21.2.18 function wc_DsaExportParamsRaw

```
int wc_DsaExportParamsRaw(
    DsaKey * dsa,
    byte * p,
    word32 * pSz,
    byte * q,
    word32 * qSz,
    byte * g,
    word32 * gSz
)
```

Exports DSA parameters to raw format.

Parameters:

- **dsa** DSA key structure
- **p** P parameter buffer
- **pSz** P parameter size (in/out)
- **q** Q parameter buffer
- **qSz** Q parameter size (in/out)
- **g** G parameter buffer
- **gSz** G parameter size (in/out)

See: wc_DsaImportParamsRaw

Return:

- 0 on success
- negative on failure

Example

```
byte p[256], q[32], g[256];
word32 pSz = sizeof(p), qSz = sizeof(q), gSz = sizeof(g);
int ret = wc_DsaExportParamsRaw(&dsa, p, &pSz, q, &qSz, g,
                                &gSz);
```

C.21.2.19 function wc_DsaExportKeyRaw

```
int wc_DsaExportKeyRaw(
    DsaKey * dsa,
    byte * x,
    word32 * xSz,
    byte * y,
    word32 * ySz
)
```

Exports DSA key to raw format.

Parameters:

- **dsa** DSA key structure
- **x** Private key buffer
- **xSz** Private key size (in/out)
- **y** Public key buffer
- **ySz** Public key size (in/out)

See: wc_DsaImportParamsRaw

Return:

- 0 on success
- negative on failure

Example

```
byte x[32], y[256];
word32 xSz = sizeof(x), ySz = sizeof(y);
int ret = wc_DsaExportKeyRaw(&dsa, x, &xSz, y, &ySz);
```

C.21.3 Source code

```
int wc_InitDsaKey(DsaKey* key);

void wc_FreeDsaKey(DsaKey* key);

int wc_DsaSign(const byte* digest, byte* out,
                           DsaKey* key, WC_RNG* rng);

int wc_DsaVerify(const byte* digest, const byte* sig,
                             DsaKey* key, int* answer);
```

```
int wc_DsaCheckPubKey(DsaKey* key);

int wc_DsaPublicKeyDecode(const byte* input, word32* inOutIdx,
                                    DsaKey* key, word32 inSz);

int wc_DsaPrivateKeyDecode(const byte* input, word32* inOutIdx,
                                     DsaKey* key, word32 inSz);

int wc_DsaKeyToDer(DsaKey* key, byte* output, word32 outLen);

int wc_MakeDsaKey(WC_RNG *rng, DsaKey *dsa);

int wc_MakeDsaParameters(WC_RNG *rng, int modulus_size, DsaKey *dsa);
int wc_InitDsaKey_h(DsaKey* key, void* h);

int wc_DsaSign_ex(const byte* digest, word32 digestSz, byte* out,
    DsaKey* key, WC_RNG* rng);

int wc_DsaVerify_ex(const byte* digest, word32 digestSz,
    const byte* sig, DsaKey* key, int* answer);

int wc_SetDsaPublicKey(byte* output, DsaKey* key, int outLen,
    int with_header);

int wc_DsaKeyToPublicDer(DsaKey* key, byte* output, word32 outLen);

int wc_DsaImportParamsRaw(DsaKey* dsa, const char* p, const char* q,
    const char* g);

int wc_DsaImportParamsRawCheck(DsaKey* dsa, const char* p,
    const char* q, const char* g, int trusted, WC_RNG* rng);

int wc_DsaExportParamsRaw(DsaKey* dsa, byte* p, word32* pSz, byte* q,
    word32* qSz, byte* g, word32* gSz);

int wc_DsaExportKeyRaw(DsaKey* dsa, byte* x, word32* xSz, byte* y,
    word32* ySz);
```

C.22 ecc.h

C.22.1 Functions

	Name
int	**wc_ecc_make_key**(WC_RNG * rng, int keysize, ecc_key * key)This function generates a new ecc_key and stores it in key.
int	**wc_ecc_make_key_ex**(WC_RNG * rng, int keysize, ecc_key * key, int curve_id)This function generates a new ecc_key and stores it in key.

	Name
int	**wc_ecc_make_pub**(ecc_key * key, ecc_point * pubOut)wc_ecc_make_pub computes the public component from an ecc_key with an existing private component. If pubOut is supplied, the computed public key is stored there, else it is stored in the supplied ecc_key public component slot.
int	**wc_ecc_make_pub_ex**(ecc_key * key, ecc_point * pubOut, WC_RNG * rng)wc_ecc_make_pub_ex computes the public component from an ecc_key with an existing private component. If pubOut is supplied, the computed public key is stored there, else it is stored in the supplied ecc_key public component slot. The supplied rng, if non-NULL, is used to blind the private key value used in the computation.
int	**wc_ecc_check_key**(ecc_key * key)Perform sanity checks on ecc key validity.
void	**wc_ecc_key_free**(ecc_key * key)This function frees an ecc_key key after it has been used.
int	**wc_ecc_shared_secret**(ecc_key * private_key, ecc_key * public_key, byte * out, word32 * outlen)This function generates a new secret key using a local private key and a received public key. It stores this shared secret key in the buffer out and updates outlen to hold the number of bytes written to the output buffer.
int	**wc_ecc_shared_secret_ex**(ecc_key * private_key, ecc_point * point, byte * out, word32 * outlen)Create an ECC shared secret between private key and public point. The point is first given full public key validation on the private key's curve with wc_ecc_check_key() (SP 800_56Ar3 5.6.2.3.3). A build without the order check (NO_ECC_CHECK_PUBKEY_ORDER) still checks range and curve.
int	**wc_ecc_sign_hash**(const byte * in, word32 inlen, byte * out, word32 * outlen, WC_RNG * rng, ecc_key * key)This function signs a message digest using an ecc_key object to guarantee authenticity.
int	**wc_ecc_sign_hash_ex**(const byte * in, word32 inlen, WC_RNG * rng, ecc_key * key, mp_int * r, mp_int * s)Sign a message digest.
int	**wc_ecc_verify_hash**(const byte * sig, word32 siglen, const byte * hash, word32 hashlen, int * res, ecc_key * key)This function verifies the ECC signature of a hash to ensure authenticity. It returns the answer through res, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

	Name
int	**wc_ecc_verify_hash_ex**(mp_int * r, mp_int * s, const byte * hash, word32 hashlen, int * res, ecc_key * key)Verify an ECC signature. Result is written to res. 1 is valid, 0 is invalid. Note: Do not use the return value to test for valid. Only use res.
int	**wc_ecc_init**(ecc_key * key)This function initializes an ecc_key object for future use with message verification or key negotiation.
int	**wc_ecc_init_ex**(ecc_key * key, void * heap, int devId)This function initializes an ecc_key object for future use with message verification or key negotiation.
ecc_key *	**wc_ecc_key_new**(void * heap)This function uses a user defined heap and allocates space for the key structure.
ecc_key *	**wc_ecc_key_new_ex**(void * heap, int devId)This function uses a user defined heap and allocates space for the key structure.
int	**wc_ecc_free**(ecc_key * key)This function frees an ecc_key object after it has been used.
void	**wc_ecc_fp_free**(void)This function frees the fixed_point cache, which can be used with ecc to speed up computation times. To use this functionality, FP_ECC (fixed_point ecc), should be defined. Threaded applications should call this function before exiting the thread.
int	**wc_ecc_is_valid_idx**(int n)Checks if an ECC idx is valid.
ecc_point *	**wc_ecc_new_point**(void)Allocate a new ECC point.
void	**wc_ecc_del_point**(ecc_point * p)Free an ECC point from memory.
int	**wc_ecc_copy_point**(const ecc_point * p, ecc_point * r)Copy the value of one point to another one.
int	**wc_ecc_cmp_point**(ecc_point * a, ecc_point * b)Compare the value of a point with another one.
int	**wc_ecc_point_is_at_infinity**(ecc_point * p)Checks if a point is at infinity. Returns 1 if point is at infinity, 0 if not, < 0 on error.
int	**wc_ecc_mulmod**(const mp_int * k, ecc_point * G, ecc_point * R, mp_int * a, mp_int * modulus, int map)Perform ECC Fixed Point multiplication.
int	**wc_ecc_export_x963**(ecc_key * key, byte * out, word32 * outLen)This function exports the ECC key from the ecc_key structure, storing the result in out. The key will be stored in ANSI X9.63 format. It stores the bytes written to the output buffer in outLen.

	Name
int	**wc_ecc_export_x963_ex**(ecc_key * key, byte * out, word32 * outLen, int compressed)This function exports the public key from the ecc_key structure, storing the result in out. The key will be stored in ANSI X9.63 format. It stores the bytes written to the output buffer in outLen. This function allows the additional option of compressing the certificate through the compressed parameter. When this parameter is true, the key will be stored in ANSI X9.63 compressed format.
int	**wc_ecc_import_x963**(const byte * in, word32 inLen, ecc_key * key)This function imports a public ECC key from a buffer containing the key stored in ANSI X9.63 format. This function will handle both compressed and uncompressed keys, as long as compressed keys are enabled at compile time through the HAVE_COMP_KEY option.
int	**wc_ecc_import_private_key**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ecc_key * key)This function imports a public/private ECC key pair from a buffer containing the raw private key, and a second buffer containing the ANSI X9.63 formatted public key. This function will handle both compressed and uncompressed keys, as long as compressed keys are enabled at compile time through the HAVE_COMP_KEY option.
int	**wc_ecc_import_wrapped_private**(ecc_key * key, int curve_id, const byte * seed, word32 seedSz, const byte * wrapped, word32 wrappedLen, word32 plainLen)This function imports an STM32 DHUK_protected private key onto an ecc_key for transparent hardware signing. The private scalar is supplied as a chip_bound wrapped blob together with the 256_bit derivation seed; the plaintext scalar is never imported. The key must be bound to the STM32 DHUK crypto_callback device (init with wc_ecc_init_ex(&key, heap, WC_DHUK_DEVID) after registering the device with wc_Stm32_DhukRegister). The curve is set from curve_id, so the key is ready to sign on return. Available only on STM32 builds with WOLFSSL_DHUK and a DHUK_capable SAES (WC_STM32_HAS_DHUK).

COPYRIGHT ©2024 wolfSSL Inc.

	Name
int	**wc_ecc_import_wrapped_private_ex**(ecc_key * key, int curve_id, const byte * wrapped, word32 wrappedLen, const byte * iv, word32 ivLen, const byte * tag, word32 tagLen, const byte * pub, word32 pubLen)This function restores a previously provisioned STM32 CCB_protected ECDSA key onto an ecc_key. The device_bound key is supplied as the wrapped scalar blob plus its AES_GCM iv/tag and the in_clear public key; signing is performed transparently with the scalar unwrapped SAES_>PKA in hardware. The key must be bound to the STM32 DHUK/CCB crypto_callback device (init with wc_ecc_init_ex(&key, heap, WC_DHUK_DEVID)). Available only on STM32 builds with WOLFSSL_DHUK and WOLFSSL_STM32_CCB.
int	**wc_ecc_rs_to_sig**(const char * r, const char * s, byte * out, word32 * outlen)This function converts the R and S portions of an ECC signature into a DER-encoded ECDSA signature. This function also stores the length written to the output buffer, out, in outlen.
int	**wc_ecc_import_raw**(ecc_key * key, const char * qx, const char * qy, const char * d, const char * curveName)This function fills an ecc_key structure with the raw components of an ECC key.
int	**wc_ecc_export_private_only**(ecc_key * key, byte * out, word32 * outLen)This function exports only the private key from an ecc_key structure. It stores the private key in the buffer out, and sets the bytes written to this buffer in outLen.
int	**wc_ecc_export_point_der**(const int curve_idx, ecc_point * point, byte * out, word32 * outLen)Export point to der.
int	**wc_ecc_import_point_der**(const byte * in, word32 inLen, const int curve_idx, ecc_point * point)Import point from der format.
int	**wc_ecc_size**(ecc_key * key)This function returns the key size of an ecc_key structure in octets.
int	**wc_ecc_sig_size_calc**(int sz)This function returns the worst case size for an ECC signature, given by: (keySz * 2) + SIG_HEADER_SZ + ECC_MAX_PAD_SZ. The actual signature size can be computed with wc_ecc_sign_hash.

	Name
int	**wc_ecc_sig_size**(const ecc_key * key)This function returns the worst case size for an ECC signature, given by: (keySz * 2) + SIG_HEADER_SZ + ECC_MAX_PAD_SZ. The actual signature size can be computed with wc_ecc_sign_hash.
ecEncCtx *	**wc_ecc_ctx_new**(int flags, WC_RNG * rng)This function allocates and initializes space for a new ECC context object to allow secure message exchange with ECC.
void	**wc_ecc_ctx_free**(ecEncCtx * ctx)This function frees the ecEncCtx object used for encrypting and decrypting messages.
int	**wc_ecc_ctx_reset**(ecEncCtx * ctx, WC_RNG * rng)This function resets an ecEncCtx structure to avoid having to free and allocate a new context object.
int	**wc_ecc_ctx_set_algo**(ecEncCtx * ctx, byte encAlgo, byte kdfAlgo, byte macAlgo)This function can optionally be called after wc_ecc_ctx_new. It sets the encryption, KDF, and MAC algorithms into an ecEncCtx object.
int	**wc_ecc_ctx_set_dev_id**(ecEncCtx * ctx, int devId)This function picks the device that ECIES operations using this context run on. Only available when WOLF_CRYPTO_CB is defined. A context starts at INVALID_DEVID, meaning software: ECIES does not copy the device from the private key, so this must be called for a crypto callback to be reached. The value is used both for the whole_operation ECIES callback and for the KDF, AES and HMAC steps of the software path. Passing a NULL context to wc_ecc_encrypt() or wc_ecc_decrypt() always means software. When WOLF_CRYPTO_CB_FIND is defined, an unset device id still goes through the registered finder, as it does for every other wolfCrypt operation. The setting is kept across wc_ecc_ctx_reset().
int	**wc_ecc_ctx_get_dev_id**(ecEncCtx * ctx, int * devId)This function reads back the device id set with wc_ecc_ctx_set_dev_id(). Crypto callback code can use it to learn which device it was called for. Only available when WOLF_CRYPTO_CB is defined. A context that was never given a device reads back INVALID_DEVID.
const byte *	**wc_ecc_ctx_get_own_salt**(ecEncCtx * ctx)This function returns the salt of an ecEncCtx object. This function should only be called when the ecEncCtx's state is ecSRV_INIT or ecCLI_INIT.

	Name
int	**wc_ecc_ctx_set_peer_salt**(ecEncCtx * ctx, const byte * salt)This function sets the peer salt of an ecEncCtx object.
int	**wc_ecc_ctx_set_kdf_salt**(ecEncCtx * ctx, const byte * salt, word32 sz)This function sets the salt pointer and length to use with KDF into the ecEncCtx object.
int	**wc_ecc_ctx_set_info**(ecEncCtx * ctx, const byte * info, int sz)This function can optionally be called before or after wc_ecc_ctx_set_peer_salt. It sets optional information for an ecEncCtx object.
int	**wc_ecc_encrypt**(ecc_key * privKey, ecc_key * pubKey, const byte * msg, word32 msgSz, byte * out, word32 * outSz, ecEncCtx * ctx)This function encrypts the given input message from msg to out. This function takes an optional ctx object as parameter. When supplied, encryption proceeds based on the ecEncCtx's encAlgo, kdfAlgo, and macAlgo. If ctx is not supplied, processing completes with the default algorithms, ecAES_128_CBC, ecHKDF_SHA256 and ecHMAC_SHA256. This function requires that the messages are padded according to the encryption type specified by ctx.
int	**wc_ecc_encrypt_ex**(ecc_key * privKey, ecc_key * pubKey, const byte * msg, word32 msgSz, byte * out, word32 * outSz, ecEncCtx * ctx, int compressed)This function encrypts the given input message from msg to out. This function takes an optional ctx object as parameter. When supplied, encryption proceeds based on the ecEncCtx's encAlgo, kdfAlgo, and macAlgo. If ctx is not supplied, processing completes with the default algorithms, ecAES_128_CBC, ecHKDF_SHA256 and ecHMAC_SHA256. This function requires that the messages are padded according to the encryption type specified by ctx.

	Name
int	**wc_ecc_decrypt**(ecc_key * privKey, ecc_key * pubKey, const byte * msg, word32 msgSz, byte * out, word32 * outSz, ecEncCtx * ctx)This function decrypts the ciphertext from msg to out. This function takes an optional ctx object as parameter. When supplied, encryption proceeds based on the ecEncCtx's encAlgo, kdfAlgo, and macAlgo. If ctx is not supplied, processing completes with the default algorithms, ecAES_128_CBC, ecHKDF_SHA256 and ecHMAC_SHA256. This function requires that the messages are padded according to the encryption type specified by ctx.
int	**wc_ecc_set_nonblock**(ecc_key * key, ecc_nb_ctx_t * ctx)Enable ECC support for non_blocking operations. Supported for Single Precision (SP) math with the following build options: WOLFSSL_SP_NONBLOCK WOLFSSL_SP_SMALL WOLFSSL_SP_NO_MALLOC WC_ECC_NONBLOCK.
int	**wc_ecc_set_curve**(ecc_key * key, int keysize, int curve_id)Compare a curve which has larger key than specified size or the curve matched curve ID, set a curve with smaller key size to the key.
mp_int *	**wc_ecc_key_get_priv**(ecc_key * key)Gets private key mp_int from ECC key.
size_t	**wc_ecc_get_sets_count**(void)Returns number of supported ECC curve sets.
const char *	**wc_ecc_get_name**(int curve_id)Gets curve name from curve ID.
int	**wc_ecc_make_key_ex2**(WC_RNG * rng, int keysize, ecc_key * key, int curve_id, int flags)Makes ECC key with extended options.
int	**wc_ecc_is_point**(ecc_point * ecp, mp_int * a, mp_int * b, mp_int * prime)Checks if point is on curve.
int	**wc_ecc_get_generator**(ecc_point * ecp, int curve_idx)Gets generator point for curve.
int	**wc_ecc_set_deterministic**(ecc_key * key, byte flag)Sets deterministic signing mode.
int	**wc_ecc_set_deterministic_ex**(ecc_key * key, byte flag, enum wc_HashType hashType)Sets deterministic signing with hash type.
int	**wc_ecc_gen_deterministic_k**(const byte * hash, word32 hashSz, enum wc_HashType hashType, mp_int * priv, mp_int * k, mp_int * order, void * heap)Generates deterministic k value for signing.
int	**wc_ecc_sign_set_k**(const byte * k, word32 klen, ecc_key * key)Sets k value for signing.

	Name
int	**wc_ecc_init_id**(ecc_key * key, unsigned char * id, int len, void * heap, int devId)Initializes ECC key with ID.
int	**wc_ecc_init_label**(ecc_key * key, const char * label, void * heap, int devId)Initializes ECC key with label.
int	**wc_ecc_set_flags**(ecc_key * key, word32 flags)Sets flags on ECC key.
void	**wc_ecc_fp_init**(void)Initializes fixed-point cache.
int	**wc_ecc_set_rng**(ecc_key * key, WC_RNG * rng)Sets RNG for ECC key.
int	**wc_ecc_get_curve_idx**(int curve_id)Gets curve index from curve ID.
int	**wc_ecc_get_curve_id**(int curve_idx)Gets curve ID from curve index.
int	**wc_ecc_get_curve_size_from_id**(int curve_id)Gets curve size from curve ID.
int	**wc_ecc_get_curve_idx_from_name**(const char * curveName)Gets curve index from curve name.
int	**wc_ecc_get_curve_size_from_name**(const char * curveName)Gets curve size from curve name.
int	**wc_ecc_get_curve_id_from_name**(const char * curveName)Gets curve ID from curve name.
int	**wc_ecc_get_curve_id_from_params**(int fieldSize, const byte * prime, word32 primeSz, const byte * Af, word32 AfSz, const byte * Bf, word32 BfSz, const byte * order, word32 orderSz, const byte * Gx, word32 GxSz, const byte * Gy, word32 GySz, int cofactor)Gets curve ID from curve parameters.
int	**wc_ecc_get_curve_id_from_dp_params**(const ecc_set_type * dp)Gets curve ID from domain parameters.
int	**wc_ecc_get_curve_id_from_oid**(const byte * oid, word32 len)Gets curve ID from OID.
const ecc_set_type *	**wc_ecc_get_curve_params**(int curve_idx)Gets curve parameters from curve index.
ecc_point *	**wc_ecc_new_point_h**(void * h)Allocates new ECC point with heap hint.
void	**wc_ecc_del_point_h**(ecc_point * p, void * h)Frees ECC point with heap hint.
void	**wc_ecc_forcezero_point**(ecc_point * p)Securely zeros ECC point.
int	**wc_ecc_point_is_on_curve**(ecc_point * p, int curve_idx)Checks if point is on curve.
int	**wc_ecc_import_x963_ex**(const byte * in, word32 inLen, ecc_key * key, int curve_id)Imports X9.63 format with curve ID.

	Name
int	**wc_ecc_import_private_key_ex**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ecc_key * key, int curve_id)Imports private key with curve ID.
int	**wc_ecc_rs_raw_to_sig**(const byte * r, word32 rSz, const byte * s, word32 sSz, byte * out, word32 * outlen)Converts raw r,s to signature.
int	**wc_ecc_sig_to_rs**(const byte * sig, word32 sigLen, byte * r, word32 * rLen, byte * s, word32 * sLen)Converts signature to raw r,s.
int	**wc_ecc_import_raw_ex**(ecc_key * key, const char * qx, const char * qy, const char * d, int curve_id)Imports raw key with curve ID.
int	**wc_ecc_import_unsigned**(ecc_key * key, const byte * qx, const byte * qy, const byte * d, int curve_id)Imports unsigned key with curve ID.
int	**wc_ecc_export_ex**(ecc_key * key, byte * qx, word32 * qxLen, byte * qy, word32 * qyLen, byte * d, word32 * dLen, int encType)Exports key with encoding type.
int	**wc_ecc_export_public_raw**(ecc_key * key, byte * qx, word32 * qxLen, byte * qy, word32 * qyLen)Exports public key in raw format.
int	**wc_ecc_export_private_raw**(ecc_key * key, byte * qx, word32 * qxLen, byte * qy, word32 * qyLen, byte * d, word32 * dLen)Exports private key in raw format.
int	**wc_ecc_export_point_der_ex**(const int curve_idx, ecc_point * point, byte * out, word32 * outLen, int compressed)Exports point in DER format with compression.
int	**wc_ecc_import_point_der_ex**(const byte * in, word32 inLen, const int curve_idx, ecc_point * point, int shortKeySize)Imports point from DER format.
int	**wc_ecc_get_oid**(word32 oidSum, const byte ** oid, word32 * oidSz)Gets OID for curve.
int	**wc_ecc_set_custom_curve**(ecc_key * key, const ecc_set_type * dp)Sets custom curve parameters.
ecEncCtx *	**wc_ecc_ctx_new_ex**(int flags, WC_RNG * rng, void * heap)Creates new ECC encryption context with heap.
int	**wc_ecc_ctx_set_own_salt**(ecEncCtx * ctx, const byte * salt, word32 sz)Sets own salt in context.
int	**wc_X963_KDF**(enum wc_HashType type, const byte * secret, word32 secretSz, const byte * sinfo, word32 sinfoSz, byte * out, word32 outSz)X9.63 Key Derivation Function.
int	**wc_ecc_curve_cache_init**(void)Initializes curve cache.

	Name
void	**wc_ecc_curve_cache_free**(void)Frees curve cache.
int	**wc_ecc_gen_k**(WC_RNG * rng, int size, mp_int * k, mp_int * order)Generates random k value.
int	**wc_ecc_set_handle**(ecc_key * key, remote_handle64 handle)Sets remote handle for hardware.
int	**wc_ecc_use_key_id**(ecc_key * key, word32 keyId, word32 flags)Uses key ID for hardware.
int	**wc_ecc_get_key_id**(ecc_key * key, word32 * keyId)Gets key ID from hardware key.

C.22.2 Functions Documentation

C.22.2.1 function wc_ecc_make_key

```
int wc_ecc_make_key(
    WC_RNG * rng,
    int keysize,
    ecc_key * key
)
```

This function generates a new ecc_key and stores it in key.

Parameters:

- **rng** pointer to an initialized RNG object with which to generate the key
- **keysize** desired length for the ecc_key
- **key** pointer to the ecc_key for which to generate a key

See:

- wc_ecc_init
- wc_ecc_shared_secret

Return:

- 0 Returned on success.
- ECC_BAD_ARG_E Returned if rng or key evaluate to NULL
- BAD_FUNC_ARG Returned if the specified key size is not in the correct range of supported keys
- MEMORY_E Returned if there is an error allocating memory while computing the ecc key
- MP_INIT_E may be returned if there is an error while computing the ecc key
- MP_READ_E may be returned if there is an error while computing the ecc key
- MP_CMP_E may be returned if there is an error while computing the ecc key
- MP_INVMOD_E may be returned if there is an error while computing the ecc key
- MP_EXPTMOD_E may be returned if there is an error while computing the ecc key
- MP_MOD_E may be returned if there is an error while computing the ecc key
- MP_MUL_E may be returned if there is an error while computing the ecc key
- MP_ADD_E may be returned if there is an error while computing the ecc key
- MP_MULMOD_E may be returned if there is an error while computing the ecc key
- MP_TO_E may be returned if there is an error while computing the ecc key
- MP_MEM may be returned if there is an error while computing the ecc key

Example

```
ecc_key key;
wc_ecc_init(&key);
```

```
WC_RNG rng;
wc_InitRng(&rng);
wc_ecc_make_key(&rng, 32, &key); // initialize 32 byte ecc key
```

C.22.2.2 function wc_ecc_make_key_ex

```
int wc_ecc_make_key_ex(
    WC_RNG * rng,
    int keysize,
    ecc_key * key,
    int curve_id
)
```

This function generates a new ecc_key and stores it in key.

Parameters:

- **key** Pointer to store the created key.
- **keysize** size of key to be created in bytes, set based on curveId
- **rng** Rng to be used in key creation
- **curve_id** Curve to use for key

See:

- wc_ecc_make_key
- wc_ecc_get_curve_size_from_id

Return:

- 0 Returned on success.
- ECC_BAD_ARG_E Returned if rng or key evaluate to NULL
- BAD_FUNC_ARG Returned if the specified key size is not in the correct range of supported keys
- MEMORY_E Returned if there is an error allocating memory while computing the ecc key
- MP_INIT_E may be returned if there is an error while computing the ecc key
- MP_READ_E may be returned if there is an error while computing the ecc key
- MP_CMP_E may be returned if there is an error while computing the ecc key
- MP_INVMOD_E may be returned if there is an error while computing the ecc key
- MP_EXPTMOD_E may be returned if there is an error while computing the ecc key
- MP_MOD_E may be returned if there is an error while computing the ecc key
- MP_MUL_E may be returned if there is an error while computing the ecc key
- MP_ADD_E may be returned if there is an error while computing the ecc key
- MP_MULMOD_E may be returned if there is an error while computing the ecc key
- MP_TO_E may be returned if there is an error while computing the ecc key
- MP_MEM may be returned if there is an error while computing the ecc key

Example

```
ecc_key key;
int ret;
WC_RNG rng;
wc_ecc_init(&key);
wc_InitRng(&rng);
int curveId = ECC_SECP521R1;
int keySize = wc_ecc_get_curve_size_from_id(curveId);
ret = wc_ecc_make_key_ex(&rng, keySize, &key, curveId);
if (ret != MP_OKAY) {
    // error handling
}
```

C.22.2.3 function wc_ecc_make_pub

```
int wc_ecc_make_pub(
    ecc_key * key,
    ecc_point * pubOut
)
```

wc_ecc_make_pub computes the public component from an ecc_key with an existing private component. If pubOut is supplied, the computed public key is stored there, else it is stored in the supplied ecc_key public component slot.

Parameters:

- **key** Pointer to an ecc_key containing a valid private component
- **pubOut** Optional pointer to an ecc_point struct in which to store the computed public key

See:

- wc_ecc_make_pub_ex
- wc_ecc_make_key

Return:

- 0 Returned on success.
- ECC_BAD_ARG_E Returned if key is NULL
- BAD_FUNC_ARG Returned if the supplied key is not a valid ecc_key.
- MEMORY_E Returned if there is an error allocating memory while computing the public key
- MP_INIT_E may be returned if there is an error while computing the public key
- MP_READ_E may be returned if there is an error while computing the public key
- MP_CMP_E may be returned if there is an error while computing the public key
- MP_INVMOD_E may be returned if there is an error while computing the public key
- MP_EXPTMOD_E may be returned if there is an error while computing the public key
- MP_MOD_E may be returned if there is an error while computing the public key
- MP_MUL_E may be returned if there is an error while computing the public key
- MP_ADD_E may be returned if there is an error while computing the public key
- MP_MULMOD_E may be returned if there is an error while computing the public key
- MP_TO_E may be returned if there is an error while computing the public key
- MP_MEM may be returned if there is an error while computing the public key
- ECC_OUT_OF_RANGE_E may be returned if there is an error while computing the public key
- ECC_PRIV_KEY_E may be returned if there is an error while computing the public key
- ECC_INF_E may be returned if there is an error while computing the public key

C.22.2.4 function wc_ecc_make_pub_ex

```
int wc_ecc_make_pub_ex(
    ecc_key * key,
    ecc_point * pubOut,
    WC_RNG * rng
)
```

wc_ecc_make_pub_ex computes the public component from an ecc_key with an existing private component. If pubOut is supplied, the computed public key is stored there, else it is stored in the supplied ecc_key public component slot. The supplied rng, if non-NULL, is used to blind the private key value used in the computation.

Parameters:

- **key** Pointer to an ecc_key containing a valid private component
- **pubOut** Optional pointer to an ecc_point struct in which to store the computed public key
- **rng** Rng to be used in the public key computation

See:

- wc_ecc_make_pub
- wc_ecc_make_key
- wc_InitRng

Return:

- 0 Returned on success.
- ECC_BAD_ARG_E Returned if key is NULL
- BAD_FUNC_ARG Returned if the supplied key is not a valid ecc_key.
- MEMORY_E Returned if there is an error allocating memory while computing the public key
- MP_INIT_E may be returned if there is an error while computing the public key
- MP_READ_E may be returned if there is an error while computing the public key
- MP_CMP_E may be returned if there is an error while computing the public key
- MP_INVMOD_E may be returned if there is an error while computing the public key
- MP_EXPTMOD_E may be returned if there is an error while computing the public key
- MP_MOD_E may be returned if there is an error while computing the public key
- MP_MUL_E may be returned if there is an error while computing the public key
- MP_ADD_E may be returned if there is an error while computing the public key
- MP_MULMOD_E may be returned if there is an error while computing the public key
- MP_TO_E may be returned if there is an error while computing the public key
- MP_MEM may be returned if there is an error while computing the public key
- ECC_OUT_OF_RANGE_E may be returned if there is an error while computing the public key
- ECC_PRIV_KEY_E may be returned if there is an error while computing the public key
- ECC_INF_E may be returned if there is an error while computing the public key

C.22.2.5 function wc_ecc_check_key

```
int wc_ecc_check_key(
    ecc_key * key
)
```

Perform sanity checks on ecc key validity.

Parameters:

- **key** Pointer to key to check.

See: wc_ecc_point_is_at_infinity

Return:

- MP_OKAY Success, key is OK.
- BAD_FUNC_ARG Returns if key is NULL.
- ECC_INF_E Returns if wc_ecc_point_is_at_infinity returns 1.

Example

```
ecc_key key;
WC_RNG rng;
int check_result;
wc_ecc_init(&key);
wc_InitRng(&rng);
wc_ecc_make_key(&rng, 32, &key);
check_result = wc_ecc_check_key(&key);

if (check_result == MP_OKAY)
{
    // key check succeeded
```

```
}
else
{
    // key check failed
}
```

C.22.2.6 function wc_ecc_key_free

```
void wc_ecc_key_free(
    ecc_key * key
)
```

This function frees an ecc_key key after it has been used.

Parameters:

- **key** pointer to the ecc_key structure to free

See:

- wc_ecc_key_new
- wc_ecc_init_ex

Example

```
// initialize key and perform ECC operations
...
wc_ecc_key_free(&key);
```

C.22.2.7 function wc_ecc_shared_secret

```
int wc_ecc_shared_secret(
    ecc_key * private_key,
    ecc_key * public_key,
    byte * out,
    word32 * outlen
)
```

This function generates a new secret key using a local private key and a received public key. It stores this shared secret key in the buffer out and updates outlen to hold the number of bytes written to the output buffer.

Parameters:

- **private_key** pointer to the ecc_key structure containing the local private key
- **public_key** pointer to the ecc_key structure containing the received public key
- **out** pointer to an output buffer in which to store the generated shared secret key
- **outlen** pointer to the word32 object containing the length of the output buffer. Will be overwritten with the length written to the output buffer upon successfully generating a shared secret key

See:

- wc_ecc_init
- wc_ecc_make_key

Return:

- 0 Returned upon successfully generating a shared secret key
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL

- ECC_BAD_ARG_E Returned if the type of the private key given as argument, private_key, is not ECC_PRIVATEKEY, or if the public and private key types (given by ecc->dp) are not equivalent
- MEMORY_E Returned if there is an error generating a new ecc point
- BUFFER_E Returned if the generated shared secret key is too long to store in the provided buffer
- MP_INIT_E may be returned if there is an error while computing the shared key
- MP_READ_E may be returned if there is an error while computing the shared key
- MP_CMP_E may be returned if there is an error while computing the shared key
- MP_INVMOD_E may be returned if there is an error while computing the shared key
- MP_EXPTMOD_E may be returned if there is an error while computing the shared key
- MP_MOD_E may be returned if there is an error while computing the shared key
- MP_MUL_E may be returned if there is an error while computing the shared key
- MP_ADD_E may be returned if there is an error while computing the shared key
- MP_MULMOD_E may be returned if there is an error while computing the shared key
- MP_TO_E may be returned if there is an error while computing the shared key
- MP_MEM may be returned if there is an error while computing the shared key
- ECC_INF_E returned when the computed shared secret is the point at infinity

Example

```
ecc_key priv, pub;
WC_RNG rng;
byte secret[1024]; // can hold 1024 byte shared secret key
word32 secretSz = sizeof(secret);
int ret;

wc_InitRng(&rng); // initialize rng
wc_ecc_init(&priv); // initialize key
wc_ecc_make_key(&rng, 32, &priv); // make public/private key pair
// receive public key, and initialise into pub
ret = wc_ecc_shared_secret(&priv, &pub, secret, &secretSz);
// generate secret key
if ( ret != 0 ) {
    // error generating shared secret key
}
```

C.22.2.8 function wc_ecc_shared_secret_ex

```
int wc_ecc_shared_secret_ex(
    ecc_key * private_key,
    ecc_point * point,
    byte * out,
    word32 * outlen
)
```

Create an ECC shared secret between private key and public point. The point is first given full public key validation on the private key's curve with wc_ecc_check_key() (SP 800_56Ar3 5.6.2.3.3). A build without the order check (NO_ECC_CHECK_PUBKEY_ORDER) still checks range and curve.

Parameters:

- **private_key** The private ECC key.
- **point** The point to use (public key).
- **out** Output destination of the shared secret. Conforms to EC-DH from ANSI X9.63.
- **outlen** Input the max size and output the resulting size of the shared secret.

See: wc_ecc_verify_hash_ex

Return:

- MP_OKAY Indicates success.
- BAD_FUNC_ARG Error returned when any arguments are null.
- ECC_BAD_ARG_E Error returned if private_key->type is not ECC_PRIVATEKEY or private_key->idx fails to validate.
- BUFFER_E Error when outlen is too small.
- MEMORY_E Error to create a new point.
- MP_VAL possible when an initialization failure occurs.
- MP_MEM possible when an initialization failure occurs.
- IS_POINT_E, ECC_OUT_OF_RANGE_E, ECC_INF_E or MP_VAL when the point fails validation (not on the curve, out of range, or the identity).

Example

```
ecc_key key;
ecc_point* point;
byte shared_secret[];
int secret_size;
int result;

point = wc_ecc_new_point();

result = wc_ecc_shared_secret_ex(&key, point,
&shared_secret, &secret_size);

if (result != MP_OKAY)
{
    // Handle error
}
```

C.22.2.9 function wc_ecc_sign_hash

```
int wc_ecc_sign_hash(
    const byte * in,
    word32 inlen,
    byte * out,
    word32 * outlen,
    WC_RNG * rng,
    ecc_key * key
)
```

This function signs a message digest using an ecc_key object to guarantee authenticity.

Parameters:

- **in** pointer to the buffer containing the message hash to sign
- **inlen** length of the message hash to sign
- **out** buffer in which to store the generated signature
- **outlen** max length of the output buffer. Will store the bytes written to out upon successfully generating a message signature
- **key** pointer to a private ECC key with which to generate the signature

See: wc_ecc_verify_hash

Return:

- 0 Returned upon successfully generating a signature for the message digest
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the output buffer is too small to store the generated signature

- ECC_BAD_ARG_E Returned if the input key is not a private key, or if the ECC OID is invalid
- RNG_FAILURE_E Returned if the rng cannot successfully generate a satisfactory key
- MP_INIT_E may be returned if there is an error while computing the message signature
- MP_READ_E may be returned if there is an error while computing the message signature
- MP_CMP_E may be returned if there is an error while computing the message signature
- MP_INVMOD_E may be returned if there is an error while computing the message signature
- MP_EXPTMOD_E may be returned if there is an error while computing the message signature
- MP_MOD_E may be returned if there is an error while computing the message signature
- MP_MUL_E may be returned if there is an error while computing the message signature
- MP_ADD_E may be returned if there is an error while computing the message signature
- MP_MULMOD_E may be returned if there is an error while computing the message signature
- MP_TO_E may be returned if there is an error while computing the message signature
- MP_MEM may be returned if there is an error while computing the message signature

Example

```
ecc_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[512]; // will hold generated signature
sigSz = sizeof(sig);
byte digest[] = { // initialize with message hash };
wc_InitRng(&rng); // initialize rng
wc_ecc_init(&key); // initialize key
wc_ecc_make_key(&rng, 32, &key); // make public/private key pair
ret = wc_ecc_sign_hash(digest, sizeof(digest), sig, &sigSz, &key);
if ( ret != 0 ) {
    // error generating message signature
}
```

C.22.2.10 function wc_ecc_sign_hash_ex

```
int wc_ecc_sign_hash_ex(
    const byte * in,
    word32 inlen,
    WC_RNG * rng,
    ecc_key * key,
    mp_int * r,
    mp_int * s
)
```

Sign a message digest.

Parameters:

- **in** The message digest to sign.
- **inlen** The length of the digest.
- **rng** Pointer to WC_RNG struct.
- **key** A private ECC key.
- **r** The destination for r component of the signature.
- **s** The destination for s component of the signature.

See: wc_ecc_verify_hash_ex

Return:

- MP_OKAY Returned upon successfully generating a signature for the message digest

- ECC_BAD_ARG_E Returned if the input key is not a private key, or if the ECC IDX is invalid, or if any of the input parameters evaluate to NULL, or if the output buffer is too small to store the generated signature
- RNG_FAILURE_E Returned if the rng cannot successfully generate a satisfactory key
- MP_INIT_E may be returned if there is an error while computing the message signature
- MP_READ_E may be returned if there is an error while computing the message signature
- MP_CMP_E may be returned if there is an error while computing the message signature
- MP_INVMOD_E may be returned if there is an error while computing the message signature
- MP_EXPTMOD_E may be returned if there is an error while computing the message signature
- MP_MOD_E may be returned if there is an error while computing the message signature
- MP_MUL_E may be returned if there is an error while computing the message signature
- MP_ADD_E may be returned if there is an error while computing the message signature
- MP_MULMOD_E may be returned if there is an error while computing the message signature
- MP_TO_E may be returned if there is an error while computing the message signature
- MP_MEM may be returned if there is an error while computing the message signature

Example

```
ecc_key key;
WC_RNG rng;
int ret, sigSz;
mp_int r; // destination for r component of signature.
mp_int s; // destination for s component of signature.

byte sig[512]; // will hold generated signature
sigSz = sizeof(sig);
byte digest[] = { initialize with message hash };
wc_InitRng(&rng); // initialize rng
wc_ecc_init(&key); // initialize key
mp_init(&r); // initialize r component
mp_init(&s); // initialize s component
wc_ecc_make_key(&rng, 32, &key); // make public/private key pair
ret = wc_ecc_sign_hash_ex(digest, sizeof(digest), &rng, &key, &r, &s);

if ( ret != MP_OKAY ) {
    // error generating message signature
}
```

C.22.2.11 function wc_ecc_verify_hash

```
int wc_ecc_verify_hash(
    const byte * sig,
    word32 siglen,
    const byte * hash,
    word32 hashlen,
    int * res,
    ecc_key * key
)
```

This function verifies the ECC signature of a hash to ensure authenticity. It returns the answer through res, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

Parameters:

- **sig** pointer to the buffer containing the signature to verify
- **siglen** length of the signature to verify
- **hash** pointer to the buffer containing the hash of the message verified

- **hashlen** length of the hash of the message verified
- **res** pointer to the result of the verification. 1 indicates the message was successfully verified
- **key** pointer to a public ECC key with which to verify the signature

See:

- wc_ecc_sign_hash
- wc_ecc_verify_hash_ex

Return:

- 0 Returned upon successfully performing the signature verification. Note: This does not mean that the signature is verified. The authenticity information is stored instead in res
- BAD_FUNC_ARG Returned any of the input parameters evaluate to NULL
- MEMORY_E Returned if there is an error allocating memory
- MP_INIT_E may be returned if there is an error while computing the message signature
- MP_READ_E may be returned if there is an error while computing the message signature
- MP_CMP_E may be returned if there is an error while computing the message signature
- MP_INVMOD_E may be returned if there is an error while computing the message signature
- MP_EXPTMOD_E may be returned if there is an error while computing the message signature
- MP_MOD_E may be returned if there is an error while computing the message signature
- MP_MUL_E may be returned if there is an error while computing the message signature
- MP_ADD_E may be returned if there is an error while computing the message signature
- MP_MULMOD_E may be returned if there is an error while computing the message signature
- MP_TO_E may be returned if there is an error while computing the message signature
- MP_MEM may be returned if there is an error while computing the message signature

Example

```
ecc_key key;
int ret, verified = 0;

byte sig[1024] { initialize with received signature };
byte digest[] = { initialize with message hash };
// initialize key with received public key
ret = wc_ecc_verify_hash(sig, sizeof(sig), digest,sizeof(digest),
&verified, &key);
if ( ret != 0 ) {
    // error performing verification
} else if ( verified == 0 ) {
    // the signature is invalid
}
```

C.22.2.12 function wc_ecc_verify_hash_ex

```
int wc_ecc_verify_hash_ex(
    mp_int * r,
    mp_int * s,
    const byte * hash,
    word32 hashlen,
    int * res,
    ecc_key * key
)
```

Verify an ECC signature. Result is written to res. 1 is valid, 0 is invalid. Note: Do not use the return value to test for valid. Only use res.

Parameters:

- **r** The signature R component to verify
- **s** The signature S component to verify
- **hash** The hash (message digest) that was signed
- **hashlen** The length of the hash (octets)
- **res** Result of signature, 1==valid, 0==invalid
- **key** The corresponding public ECC key

See: wc_ecc_verify_hash

Return:

- MP_OKAY If successful (even if the signature is not valid)
- ECC_BAD_ARG_E Returns if arguments are null or if key-idx is invalid.
- MEMORY_E Error allocating ints or points.

Example

```
mp_int r;
mp_int s;
int res;
byte hash[] = { Some hash }
ecc_key key;

if(wc_ecc_verify_hash_ex(&r, &s, hash, hashlen, &res, &key) == MP_OKAY)
{
    // Check res
}
```

C.22.2.13 function wc_ecc_init

```
int wc_ecc_init(
    ecc_key * key
)
```

This function initializes an ecc_key object for future use with message verification or key negotiation.

Parameters:

- **key** pointer to the ecc_key object to initialize

See:

- wc_ecc_make_key
- wc_ecc_free

Return:

- 0 Returned upon successfully initializing the ecc_key object
- MEMORY_E Returned if there is an error allocating memory

Example

```
ecc_key key;
wc_ecc_init(&key);
```

C.22.2.14 function wc_ecc_init_ex

```
int wc_ecc_init_ex(
    ecc_key * key,
    void * heap,
```

```
    int devId
)
```

This function initializes an ecc_key object for future use with message verification or key negotiation.

Parameters:

- **key** pointer to the ecc_key object to initialize
- **heap** pointer to a heap identifier
- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID if not used

See:

- wc_ecc_make_key
- wc_ecc_free
- wc_ecc_init

Return:

- 0 Returned upon successfully initializing the ecc_key object
- MEMORY_E Returned if there is an error allocating memory

Example

```
ecc_key key;
wc_ecc_init_ex(&key, heap, devId);
```

C.22.2.15 function wc_ecc_key_new

```
ecc_key * wc_ecc_key_new(
    void * heap
)
```

This function uses a user defined heap and allocates space for the key structure.

Parameters:

- **heap** pointer to a heap identifier

See:

- wc_ecc_key_new_ex
- wc_ecc_make_key
- wc_ecc_key_free
- wc_ecc_init

Return:

- Non-null returned upon successfully allocating and initializing the ecc_key object
- NULL returned if there is an error allocating memory

Example

```
wc_ecc_key_new(&heap);
```

C.22.2.16 function wc_ecc_key_new_ex

```
ecc_key * wc_ecc_key_new_ex(
    void * heap,
    int devId
)
```

This function uses a user defined heap and allocates space for the key structure.

Parameters:

- **heap** pointer to a heap identifier
- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID if not used

See:

- wc_ecc_key_new
- wc_ecc_make_key
- wc_ecc_key_free
- wc_ecc_init

Return:

- Non-null returned upon successfully allocating and initializing the ecc_key object
- NULL returned if there is an error allocating memory

Example

```
wc_ecc_key_new_ex(&heap, MY_DEVID);
```

C.22.2.17 function wc_ecc_free

```
int wc_ecc_free(
    ecc_key * key
)
```

This function frees an ecc_key object after it has been used.

Parameters:

- **key** pointer to the ecc_key object to free

See: wc_ecc_init

Return: int integer returned indicating wolfSSL error or success status.

Example

```
// initialize key and perform secure exchanges
...
wc_ecc_free(&key);
```

C.22.2.18 function wc_ecc_fp_free

```
void wc_ecc_fp_free(
    void
)
```

This function frees the fixed-point cache, which can be used with ecc to speed up computation times. To use this functionality, FP_ECC (fixed-point ecc), should be defined. Threaded applications should call this function before exiting the thread.

Parameters:

- **none** No parameters.

See: wc_ecc_free

Return: none No returns.

Example

```
ecc_key key;
// initialize key and perform secure exchanges
...

wc_ecc_fp_free();
```

C.22.2.19 function wc_ecc_is_valid_idx

```
int wc_ecc_is_valid_idx(
    int n
)
```

Checks if an ECC idx is valid.

Parameters:

- **n** The idx number to check.

See: none

Return:

- 1 Return if valid.
- 0 Return if not valid.

Example

```
ecc_key key;
WC_RNG rng;
int is_valid;
wc_ecc_init(&key);
wc_InitRng(&rng);
wc_ecc_make_key(&rng, 32, &key);
is_valid = wc_ecc_is_valid_idx(key.idx);
if (is_valid == 1)
{
    // idx is valid
}
else if (is_valid == 0)
{
    // idx is not valid
}
```

C.22.2.20 function wc_ecc_new_point

```
ecc_point * wc_ecc_new_point(
    void
)
```

Allocate a new ECC point.

Parameters:

- **none** No parameters.

See:

- wc_ecc_del_point
- wc_ecc_cmp_point
- wc_ecc_copy_point
- wc_ecc_del_point

Return:

- p A newly allocated point.
- NULL Returns NULL on error.
- ecc_point pointer on success
- NULL on failure

Allocates new ECC point.

Example

```
ecc_point* point;
point = wc_ecc_new_point();
if (point == NULL)
{
    // Handle point creation error
}
// Do stuff with point
```

Example

```
ecc_point* point = wc_ecc_new_point();
if (point != NULL) {
    // use point
    wc_ecc_del_point(point);
}
```

C.22.2.21 function wc_ecc_del_point

```
void wc_ecc_del_point(
    ecc_point * p
)
```

Free an ECC point from memory.

Parameters:

- **p** The point to free.

See:

- wc_ecc_new_point
- wc_ecc_cmp_point
- wc_ecc_copy_point

Return: none No returns.

Example

```
ecc_point* point;
point = wc_ecc_new_point();
if (point == NULL)
{
    // Handle point creation error
}
// Do stuff with point
wc_ecc_del_point(point);
```

C.22.2.22 function wc_ecc_copy_point

```
int wc_ecc_copy_point(
    const ecc_point * p,
    ecc_point * r
)
```

Copy the value of one point to another one.

Parameters:

- **p** The point to copy.
- **r** The created point.

See:

- wc_ecc_new_point
- wc_ecc_cmp_point
- wc_ecc_del_point

Return:

- ECC_BAD_ARG_E Error thrown when p or r is null.
- MP_OKAY Point copied successfully
- ret Error from internal functions. Can be...

Example

```
ecc_point* point;
ecc_point* copied_point;
int copy_return;

point = wc_ecc_new_point();
copy_return = wc_ecc_copy_point(point, copied_point);
if (copy_return != MP_OKAY)
{
    // Handle error
}
```

C.22.2.23 function wc_ecc_cmp_point

```
int wc_ecc_cmp_point(
    ecc_point * a,
    ecc_point * b
)
```

Compare the value of a point with another one.

Parameters:

- **a** First point to compare.
- **b** Second point to compare.

See:

- wc_ecc_new_point
- wc_ecc_del_point
- wc_ecc_copy_point

Return:

- BAD_FUNC_ARG One or both arguments are NULL.
- MP_EQ The points are equal.
- ret Either MP_LT or MP_GT and signifies that the points are not equal.

Example

```
ecc_point* point;
ecc_point* point_to_compare;
int cmp_result;

point = wc_ecc_new_point();
point_to_compare = wc_ecc_new_point();
cmp_result = wc_ecc_cmp_point(point, point_to_compare);
if (cmp_result == BAD_FUNC_ARG)
{
    // arguments are invalid
}
else if (cmp_result == MP_EQ)
{
    // Points are equal
}
else
{
    // Points are not equal
}
```

C.22.2.24 function wc_ecc_point_is_at_infinity

```
int wc_ecc_point_is_at_infinity(
    ecc_point * p
)
```

Checks if a point is at infinity. Returns 1 if point is at infinity, 0 if not, < 0 on error.

Parameters:

- **p** The point to check.

See:

- wc_ecc_new_point
- wc_ecc_del_point
- wc_ecc_cmp_point
- wc_ecc_copy_point

Return:

- 1 p is at infinity.
- 0 p is not at infinity.
- <0 Error.

Example

```
ecc_point* point;
int is_infinity;
point = wc_ecc_new_point();

is_infinity = wc_ecc_point_is_at_infinity(point);
if (is_infinity < 0)
{
    // Handle error
}
else if (is_infinity == 0)
```

```
{
    // Point is not at infinity
}
else if (is_infinity == 1)
{
    // Point is at infinity
}
```

C.22.2.25 function wc_ecc_mulmod

```
int wc_ecc_mulmod(
    const mp_int * k,
    ecc_point * G,
    ecc_point * R,
    mp_int * a,
    mp_int * modulus,
    int map
)
```

Perform ECC Fixed Point multiplication.

Parameters:

- **k** The multiplicand.
- **G** Base point to multiply.
- **R** Destination of product.
- **a** ECC curve parameter a.
- **modulus** The modulus for the curve.
- **map** If non-zero maps the point back to affine coordinates, otherwise it's left in jacobian-montgomery form.

See: none

Return:

- MP_OKAY Returns on successful operation.
- MP_INIT_E Returned if there is an error initializing an integer for use with the multiple precision integer (mp_int) library.

Example

```
ecc_point* base;
ecc_point* destination;
// Initialize points
base = wc_ecc_new_point();
destination = wc_ecc_new_point();
// Setup other arguments
mp_int multiplicand;
mp_int modulus;
mp_int a;
int map;
int rc;
rc = wc_ecc_mulmod(&multiplicand, base, destination, &a, &modulus, map);
```

C.22.2.26 function wc_ecc_export_x963

```
int wc_ecc_export_x963(
    ecc_key * key,
```

```
    byte * out,
    word32 * outLen
)
```

This function exports the ECC key from the ecc_key structure, storing the result in out. The key will be stored in ANSI X9.63 format. It stores the bytes written to the output buffer in outLen.

Parameters:

- **key** pointer to the ecc_key object to export
- **out** pointer to the buffer in which to store the ANSI X9.63 formatted key
- **outLen** size of the output buffer. On successfully storing the key, will hold the bytes written to the output buffer

See:

- wc_ecc_export_x963_ex
- wc_ecc_import_x963
- wc_ecc_make_pub

Return:

- 0 Returned on successfully exporting the ecc_key
- LENGTH_ONLY_E Returned if the output buffer evaluates to NULL, but the other two input parameters are valid. Indicates that the function is only returning the length required to store the key
- ECC_BAD_ARG_E Returned if any of the input parameters are NULL, or the key is unsupported (has an invalid index)
- BUFFER_E Returned if the output buffer is too small to store the ecc key. If the output buffer is too small, the size needed will be returned in outLen
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- MP_INIT_E may be returned if there is an error processing the ecc_key
- MP_READ_E may be returned if there is an error processing the ecc_key
- MP_CMP_E may be returned if there is an error processing the ecc_key
- MP_INVMOD_E may be returned if there is an error processing the ecc_key
- MP_EXPTMOD_E may be returned if there is an error processing the ecc_key
- MP_MOD_E may be returned if there is an error processing the ecc_key
- MP_MUL_E may be returned if there is an error processing the ecc_key
- MP_ADD_E may be returned if there is an error processing the ecc_key
- MP_MULMOD_E may be returned if there is an error processing the ecc_key
- MP_TO_E may be returned if there is an error processing the ecc_key
- MP_MEM may be returned if there is an error processing the ecc_key

Example

```
int ret;
byte buff[1024];
word32 buffSz = sizeof(buff);

ecc_key key;
// initialize key, make key
ret = wc_ecc_export_x963(&key, buff, &buffSz);
if ( ret != 0) {
    // error exporting key
}
```

C.22.2.27 function wc_ecc_export_x963_ex

```
int wc_ecc_export_x963_ex(
    ecc_key * key,
    byte * out,
    word32 * outLen,
    int compressed
)
```

This function exports the public key from the ecc_key structure, storing the result in out. The key will be stored in ANSI X9.63 format. It stores the bytes written to the output buffer in outLen. This function allows the additional option of compressing the certificate through the compressed parameter. When this parameter is true, the key will be stored in ANSI X9.63 compressed format.

Parameters:

- **key** pointer to the ecc_key object to export
- **out** pointer to the buffer in which to store the ANSI X9.63 formatted public key
- **outLen** size of the output buffer. On successfully storing the public key, will hold the bytes written to the output buffer
- **compressed** indicator of whether to store the key in compressed format. 1==compressed, 0==uncompressed

See:

- wc_ecc_export_x963
- wc_ecc_import_x963
- wc_ecc_make_pub

Return:

- 0 Returned on successfully exporting the ecc_key public component
- ECC_PRIVATEKEY_ONLY Returned if the ecc_key public component is missing
- NOT_COMPILED_IN Returned if the HAVE_COMP_KEY was not enabled at compile time, but the key was requested in compressed format
- LENGTH_ONLY_E Returned if the output buffer evaluates to NULL, but the other two input parameters are valid. Indicates that the function is only returning the length required to store the public key
- ECC_BAD_ARG_E Returned if any of the input parameters are NULL, or the key is unsupported (has an invalid index)
- BUFFER_E Returned if the output buffer is too small to store the public key. If the output buffer is too small, the size needed will be returned in outLen
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- MP_INIT_E may be returned if there is an error processing the ecc_key
- MP_READ_E may be returned if there is an error processing the ecc_key
- MP_CMP_E may be returned if there is an error processing the ecc_key
- MP_INVMOD_E may be returned if there is an error processing the ecc_key
- MP_EXPTMOD_E may be returned if there is an error processing the ecc_key
- MP_MOD_E may be returned if there is an error processing the ecc_key
- MP_MUL_E may be returned if there is an error processing the ecc_key
- MP_ADD_E may be returned if there is an error processing the ecc_key
- MP_MULMOD_E may be returned if there is an error processing the ecc_key
- MP_TO_E may be returned if there is an error processing the ecc_key
- MP_MEM may be returned if there is an error processing the ecc_key

Example

```
int ret;
byte buff[1024];
word32 buffSz = sizeof(buff);
ecc_key key;
```

```
// initialize key, make key
ret = wc_ecc_export_x963_ex(&key, buff, &buffSz, 1);
if ( ret != 0) {
    // error exporting key
}
```

C.22.2.28 function wc_ecc_import_x963

```
int wc_ecc_import_x963(
    const byte * in,
    word32 inLen,
    ecc_key * key
)
```

This function imports a public ECC key from a buffer containing the key stored in ANSI X9.63 format. This function will handle both compressed and uncompressed keys, as long as compressed keys are enabled at compile time through the HAVE_COMP_KEY option.

Parameters:

- **in** pointer to the buffer containing the ANSI x9.63 formatted ECC key
- **inLen** length of the input buffer
- **key** pointer to the ecc_key object in which to store the imported key

See:

- wc_ecc_export_x963
- wc_ecc_import_private_key

Return:

- 0 Returned on successfully importing the ecc_key
- NOT_COMPILED_IN Returned if the HAVE_COMP_KEY was not enabled at compile time, but the key is stored in compressed format
- ECC_BAD_ARG_E Returned if in or key evaluate to NULL, or the inLen is even (according to the x9.63 standard, the key must be odd)
- MEMORY_E Returned if there is an error allocating memory
- ASN_PARSE_E Returned if there is an error parsing the ECC key; may indicate that the ECC key is not stored in valid ANSI X9.63 format
- IS_POINT_E Returned if the public key exported is not a point on the ECC curve
- MP_INIT_E may be returned if there is an error processing the ecc_key
- MP_READ_E may be returned if there is an error processing the ecc_key
- MP_CMP_E may be returned if there is an error processing the ecc_key
- MP_INVMOD_E may be returned if there is an error processing the ecc_key
- MP_EXPTMOD_E may be returned if there is an error processing the ecc_key
- MP_MOD_E may be returned if there is an error processing the ecc_key
- MP_MUL_E may be returned if there is an error processing the ecc_key
- MP_ADD_E may be returned if there is an error processing the ecc_key
- MP_MULMOD_E may be returned if there is an error processing the ecc_key
- MP_TO_E may be returned if there is an error processing the ecc_key
- MP_MEM may be returned if there is an error processing the ecc_key

Example

```
int ret;
byte buff[] = { initialize with ANSI X9.63 formatted key };

ecc_key pubKey;
wc_ecc_init(&pubKey);
```

```
ret = wc_ecc_import_x963(buff, sizeof(buff), &pubKey);
if ( ret != 0) {
    // error importing key
}
```

C.22.2.29 function wc_ecc_import_private_key

```
int wc_ecc_import_private_key(
    const byte * priv,
    word32 privSz,
    const byte * pub,
    word32 pubSz,
    ecc_key * key
)
```

This function imports a public/private ECC key pair from a buffer containing the raw private key, and a second buffer containing the ANSI X9.63 formatted public key. This function will handle both compressed and uncompressed keys, as long as compressed keys are enabled at compile time through the HAVE_COMP_KEY option.

Parameters:

- **priv** pointer to the buffer containing the raw private key
- **privSz** size of the private key buffer
- **pub** pointer to the buffer containing the ANSI x9.63 formatted ECC public key
- **pubSz** length of the public key input buffer
- **key** pointer to the ecc_key object in which to store the imported private/public key pair

See:

- wc_ecc_export_x963
- wc_ecc_import_private_key

Return:

- 0 Returned on successfully importing the ecc_key NOT_COMPILED_IN Returned if the HAVE_COMP_KEY was not enabled at compile time, but the key is stored in compressed format
- ECC_BAD_ARG_E Returned if in or key evaluate to NULL, or the inLen is even (according to the x9.63 standard, the key must be odd)
- MEMORY_E Returned if there is an error allocating memory
- ASN_PARSE_E Returned if there is an error parsing the ECC key; may indicate that the ECC key is not stored in valid ANSI X9.63 format
- IS_POINT_E Returned if the public key exported is not a point on the ECC curve
- MP_INIT_E may be returned if there is an error processing the ecc_key
- MP_READ_E may be returned if there is an error processing the ecc_key
- MP_CMP_E may be returned if there is an error processing the ecc_key
- MP_INVMOD_E may be returned if there is an error processing the ecc_key
- MP_EXPTMOD_E may be returned if there is an error processing the ecc_key
- MP_MOD_E may be returned if there is an error processing the ecc_key
- MP_MUL_E may be returned if there is an error processing the ecc_key
- MP_ADD_E may be returned if there is an error processing the ecc_key
- MP_MULMOD_E may be returned if there is an error processing the ecc_key
- MP_TO_E may be returned if there is an error processing the ecc_key
- MP_MEM may be returned if there is an error processing the ecc_key

Example

```
int ret;
byte pub[] = { initialize with ANSI X9.63 formatted key };
byte priv[] = { initialize with the raw private key };

ecc_key key;
wc_ecc_init(&key);
ret = wc_ecc_import_private_key(priv, sizeof(priv), pub, sizeof(pub),
&key);
if ( ret != 0) {
    // error importing key
}
```

C.22.2.30 function wc_ecc_import_wrapped_private

```
int wc_ecc_import_wrapped_private(
    ecc_key * key,
    int curve_id,
    const byte * seed,
    word32 seedSz,
    const byte * wrapped,
    word32 wrappedLen,
    word32 plainLen
)
```

This function imports an STM32 DHUK-protected private key onto an ecc_key for transparent hardware signing. The private scalar is supplied as a chip-bound wrapped blob together with the 256-bit derivation seed; the plaintext scalar is never imported. The key must be bound to the STM32 DHUK crypto-callback device (init with wc_ecc_init_ex(&key, heap, WC_DHUK_DEVID) after registering the device with wc_Stm32_DhukRegister). The curve is set from curve_id, so the key is ready to sign on return. Available only on STM32 builds with WOLFSSL_DHUK and a DHUK-capable SAES (WC_STM32_HAS_DHUK).

Parameters:

- **key** pointer to the ecc_key (bound to WC_DHUK_DEVID) to import into.
- **curve_id** curve the scalar belongs to, e.g. ECC_SECP256R1.
- **seed** pointer to the 256-bit (32-byte) per-key DHUK derivation seed.
- **seedSz** length of seed in bytes; must be 32.
- **wrapped** pointer to the DHUK-wrapped private scalar blob.
- **wrappedLen** length of the wrapped blob; a non-zero multiple of the AES block size, no larger than the on-key buffer.
- **plainLen** length in bytes of the plaintext scalar inside the blob; must equal the scalar size of curve_id (32 for P-256, 48 for P-384).

See:

- wc_ecc_import_wrapped_private_ex
- wc_ecc_sign_hash
- wc_ecc_init_ex

Return:

- 0 Returned on success.
- BAD_FUNC_ARG Returned if key, seed, or wrapped is NULL; if seedSz is not 32; if wrappedLen is zero or not a multiple of the AES block size; if wrappedLen exceeds the on-key blob buffer; if plainLen is zero or larger than wrappedLen; if wrappedLen is larger than plainLen padded to a full AES block; or if plainLen does not match the scalar size of curve_id.
- <0 An error from the curve lookup if curve_id is not supported.

Example

```
ecc_key key;
wc_Stm32_DhukRegister(WC_DHUK_DEVID);
wc_ecc_init_ex(&key, NULL, WC_DHUK_DEVID);
if (wc_ecc_import_wrapped_private(&key, ECC_SECP256R1, seed, 32, wrapped,
        wrappedLen, plainLen) == 0) {
    wc_ecc_sign_hash(hash, hashLen, sig, &sigLen, &rng, &key);
}
wc_ecc_free(&key);
```

C.22.2.31 function wc_ecc_import_wrapped_private_ex

```
int wc_ecc_import_wrapped_private_ex(
    ecc_key * key,
    int curve_id,
    const byte * wrapped,
    word32 wrappedLen,
    const byte * iv,
    word32 ivLen,
    const byte * tag,
    word32 tagLen,
    const byte * pub,
    word32 pubLen
)
```

This function restores a previously provisioned STM32 CCB-protected ECDSA key onto an ecc_key. The device-bound key is supplied as the wrapped scalar blob plus its AES-GCM iv/tag and the in-clear public key; signing is performed transparently with the scalar unwrapped SAES->PKA in hardware. The key must be bound to the STM32 DHUK/CCB crypto-callback device (init with wc_ecc_init_ex(&key, heap, WC_DHUK_DEVID)). Available only on STM32 builds with WOLFSSL_DHUK and WOLFSSL_STM32_CCB.

Parameters:

- **key** pointer to the ecc_key (bound to WC_DHUK_DEVID) to import into.
- **curve_id** the ECC curve id of the wrapped key (e.g. ECC_SECP256R1).
- **wrapped** pointer to the CCB wrapped private scalar blob.
- **wrappedLen** length of the wrapped blob, no larger than the on-key buffer.
- **iv** pointer to the 16-byte AES-GCM iv of the blob.
- **ivLen** length of iv in bytes; must be 16.
- **tag** pointer to the 16-byte AES-GCM authentication tag of the blob.
- **tagLen** length of tag in bytes; must be 16.
- **pub** pointer to the public key in uncompressed qx||qy form.
- **pubLen** length of pub in bytes; must be twice the curve modulus size.

See:

- wc_ecc_import_wrapped_private
- wc_ecc_make_key_ex
- wc_ecc_sign_hash

Return:

- 0 Returned on success.
- BAD_FUNC_ARG Returned if key, wrapped, iv, tag, or pub is NULL; if ivLen or tagLen is not 16; if curve_id is not a supported curve; if wrappedLen is zero or exceeds the on-key blob buffer; or if pubLen is not twice the curve modulus size.
- <0 A negative error code may be returned if importing the public key fails.

Example

```
ecc_key key;
wc_Stm32_DhukRegister(WC_DHUK_DEVID);
wc_ecc_init_ex(&key, NULL, WC_DHUK_DEVID);
if (wc_ecc_import_wrapped_private_ex(&key, ECC_SECP256R1, wrapped,
        wrappedLen, iv, 16, tag, 16, pub, pubLen) == 0) {
    wc_ecc_sign_hash(hash, hashLen, sig, &sigLen, &rng, &key);
}
wc_ecc_free(&key);
```

C.22.2.32 function wc_ecc_rs_to_sig

```
int wc_ecc_rs_to_sig(
    const char * r,
    const char * s,
    byte * out,
    word32 * outlen
)
```

This function converts the R and S portions of an ECC signature into a DER-encoded ECDSA signature. This function also stores the length written to the output buffer, out, in outlen.

Parameters:

- **r** pointer to the buffer containing the R portion of the signature as a string
- **s** pointer to the buffer containing the S portion of the signature as a string
- **out** pointer to the buffer in which to store the DER-encoded ECDSA signature
- **outlen** length of the output buffer available. Will store the bytes written to the buffer after successfully converting the signature to ECDSA format

See:

- wc_ecc_sign_hash
- wc_ecc_sig_size

Return:

- 0 Returned on successfully converting the signature
- ECC_BAD_ARG_E Returned if any of the input parameters evaluate to NULL, or if the input buffer is not large enough to hold the DER-encoded ECDSA signature
- MP_INIT_E may be returned if there is an error processing the ecc_key
- MP_READ_E may be returned if there is an error processing the ecc_key
- MP_CMP_E may be returned if there is an error processing the ecc_key
- MP_INVMOD_E may be returned if there is an error processing the ecc_key
- MP_EXPTMOD_E may be returned if there is an error processing the ecc_key
- MP_MOD_E may be returned if there is an error processing the ecc_key
- MP_MUL_E may be returned if there is an error processing the ecc_key
- MP_ADD_E may be returned if there is an error processing the ecc_key
- MP_MULMOD_E may be returned if there is an error processing the ecc_key
- MP_TO_E may be returned if there is an error processing the ecc_key
- MP_MEM may be returned if there is an error processing the ecc_key

Example

```
int ret;
ecc_key key;
// initialize key, generate R and S
```

```
char r[] = { initialize with R };
char s[] = { initialize with S };
byte sig[wc_ecc_sig_size(key)];
// signature size will be 2 * ECC key size + ~10 bytes for ASN.1 overhead
word32 sigSz = sizeof(sig);
ret = wc_ecc_rs_to_sig(r, s, sig, &sigSz);
if ( ret != 0) {
    // error converting parameters to signature
}
```

C.22.2.33 function wc_ecc_import_raw

```
int wc_ecc_import_raw(
    ecc_key * key,
    const char * qx,
    const char * qy,
    const char * d,
    const char * curveName
)
```

This function fills an ecc_key structure with the raw components of an ECC key.

Parameters:

- **key** pointer to an ecc_key structure to fill
- **qx** pointer to a buffer containing the x component of the base point as an ASCII hex string
- **qy** pointer to a buffer containing the y component of the base point as an ASCII hex string
- **d** pointer to a buffer containing the private key as an ASCII hex string
- **curveName** pointer to a string containing the ECC curve name, as found in ecc_sets

See: wc_ecc_import_private_key

Return:

- 0 Returned upon successfully importing into the ecc_key structure
- ECC_BAD_ARG_E Returned if any of the input values evaluate to NULL
- MEMORY_E Returned if there is an error initializing space to store the parameters of the ecc_key
- ASN_PARSE_E Returned if the input curveName is not defined in ecc_sets
- MP_INIT_E may be returned if there is an error processing the input parameters
- MP_READ_E may be returned if there is an error processing the input parameters
- MP_CMP_E may be returned if there is an error processing the input parameters
- MP_INVMOD_E may be returned if there is an error processing the input parameters
- MP_EXPTMOD_E may be returned if there is an error processing the input parameters
- MP_MOD_E may be returned if there is an error processing the input parameters
- MP_MUL_E may be returned if there is an error processing the input parameters
- MP_ADD_E may be returned if there is an error processing the input parameters
- MP_MULMOD_E may be returned if there is an error processing the input parameters
- MP_TO_E may be returned if there is an error processing the input parameters
- MP_MEM may be returned if there is an error processing the input parameters

Note: This function does not check that the imported public point lies on the curve. Define WOLFSSL_VALIDATE_ECC_IMPORT to validate the point on import, or call wc_ecc_check_key before the key is used.

Example

```
int ret;
ecc_key key;
wc_ecc_init(&key);
```

```
char qx[] = { initialize with x component of base point };
char qy[] = { initialize with y component of base point };
char d[]  = { initialize with private key };
ret = wc_ecc_import_raw(&key,qx, qy, d, "ECC-256");
if ( ret != 0) {
    // error initializing key with given inputs
}
```

C.22.2.34 function wc_ecc_export_private_only

```
int wc_ecc_export_private_only(
    ecc_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports only the private key from an ecc_key structure. It stores the private key in the buffer out, and sets the bytes written to this buffer in outLen.

Parameters:

- **key** pointer to an ecc_key structure from which to export the private key
- **out** pointer to the buffer in which to store the private key
- **outLen** pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the private key

See: wc_ecc_import_private_key

Return:

- 0 Returned upon successfully exporting the private key
- ECC_BAD_ARG_E Returned if any of the input values evaluate to NULL
- MEMORY_E Returned if there is an error initializing space to store the parameters of the ecc_key
- ASN_PARSE_E Returned if the input curveName is not defined in ecc_sets
- MP_INIT_E may be returned if there is an error processing the input parameters
- MP_READ_E may be returned if there is an error processing the input parameters
- MP_CMP_E may be returned if there is an error processing the input parameters
- MP_INVMOD_E may be returned if there is an error processing the input parameters
- MP_EXPTMOD_E may be returned if there is an error processing the input parameters
- MP_MOD_E may be returned if there is an error processing the input parameters
- MP_MUL_E may be returned if there is an error processing the input parameters
- MP_ADD_E may be returned if there is an error processing the input parameters
- MP_MULMOD_E may be returned if there is an error processing the input parameters
- MP_TO_E may be returned if there is an error processing the input parameters
- MP_MEM may be returned if there is an error processing the input parameters

Example

```
int ret;
ecc_key key;
// initialize key, make key

char priv[ECC_KEY_SIZE];
word32 privSz = sizeof(priv);
ret = wc_ecc_export_private_only(&key, priv, &privSz);
if ( ret != 0) {
```

```
    // error exporting private key
}
```

C.22.2.35 function wc_ecc_export_point_der

```
int wc_ecc_export_point_der(
    const int curve_idx,
    ecc_point * point,
    byte * out,
    word32 * outLen
)
```

Export point to der.

Parameters:

- **curve_idx** Index of the curve used from ecc_sets.
- **point** Point to export to der.
- **out** Destination for the output.
- **outLen** Maxsize allowed for output, destination for final size of output

See: wc_ecc_import_point_der

Return:

- 0 Returned on success.
- ECC_BAD_ARG_E Returns if curve_idx is less than 0 or invalid. Also returns when
- LENGTH_ONLY_E outLen is set but nothing else.
- BUFFER_E Returns if outLen is less than 1 + 2 * the curve size.
- MEMORY_E Returns if there is a problem allocating memory.

Example

```
int curve_idx;
ecc_point* point;
byte out[];
word32 outLen;
wc_ecc_export_point_der(curve_idx, point, out, &outLen);
```

C.22.2.36 function wc_ecc_import_point_der

```
int wc_ecc_import_point_der(
    const byte * in,
    word32 inLen,
    const int curve_idx,
    ecc_point * point
)
```

Import point from der format.

Parameters:

- **in** der buffer to import point from.
- **inLen** Length of der buffer.
- **curve_idx** Index of curve.
- **point** Destination for point.

See: wc_ecc_export_point_der

Return:

- ECC_BAD_ARG_E Returns if any arguments are null or if inLen is even.
- MEMORY_E Returns if there is an error initializing
- NOT_COMPILED_IN Returned if HAVE_COMP_KEY is not true and in is a compressed cert
- MP_OKAY Successful operation.

Example

```
byte in[];
word32 inLen;
int curve_idx;
ecc_point* point;
wc_ecc_import_point_der(in, inLen, curve_idx, point);
```

C.22.2.37 function wc_ecc_size

```
int wc_ecc_size(
    ecc_key * key
)
```

This function returns the key size of an ecc_key structure in octets.

Parameters:

- **key** pointer to an ecc_key structure for which to get the key size

See: wc_ecc_make_key

Return:

- Given a valid key, returns the key size in octets
- 0 Returned if the given key is NULL

Example

```
int keySz;
ecc_key key;
// initialize key, make key
keySz = wc_ecc_size(&key);
if ( keySz == 0) {
    // error determining key size
}
```

C.22.2.38 function wc_ecc_sig_size_calc

```
int wc_ecc_sig_size_calc(
    int sz
)
```

This function returns the worst case size for an ECC signature, given by: (keySz * 2) + SIG_HEADER_SZ + ECC_MAX_PAD_SZ. The actual signature size can be computed with wc_ecc_sign_hash.

Parameters:

- **key** size

See:

- wc_ecc_sign_hash
- wc_ecc_sig_size

Return: returns the maximum signature size, in octets

Example

```
int sigSz = wc_ecc_sig_size_calc(32);
if ( sigSz == 0) {
    // error determining sig size
}
```

C.22.2.39 function wc_ecc_sig_size

```
int wc_ecc_sig_size(
    const ecc_key * key
)
```

This function returns the worst case size for an ECC signature, given by: (keySz * 2) + SIG_HEADER_SZ + ECC_MAX_PAD_SZ. The actual signature size can be computed with wc_ecc_sign_hash.

Parameters:

- **key** pointer to an ecc_key structure for which to get the signature size

See:

- wc_ecc_sign_hash
- wc_ecc_sig_size_calc

Return:

- Success Given a valid key, returns the maximum signature size, in octets
- 0 Returned if the given key is NULL

Example

```
int sigSz;
ecc_key key;
// initialize key, make key

sigSz = wc_ecc_sig_size(&key);
if ( sigSz == 0) {
    // error determining sig size
}
```

C.22.2.40 function wc_ecc_ctx_new

```
ecEncCtx * wc_ecc_ctx_new(
    int flags,
    WC_RNG * rng
)
```

This function allocates and initializes space for a new ECC context object to allow secure message exchange with ECC.

Parameters:

- **flags** indicate whether this is a server or client context Options are: REQ_RESP_CLIENT, and REQ_RESP_SERVER
- **rng** pointer to a RNG object with which to generate a salt
- **flags** Context flags
- **rng** Random number generator

See:

- wc_ecc_encrypt
- wc_ecc_encrypt_ex

- wc_ecc_decrypt
- wc_ecc_ctx_free

Return:

- Success On successfully generating a new ecEncCtx object, returns a pointer to that object
- NULL Returned if the function fails to generate a new ecEncCtx object
- ecEncCtx pointer on success
- NULL on failure

Creates new ECC encryption context.

Example

```
ecEncCtx* ctx;
WC_RNG rng;
wc_InitRng(&rng);
ctx = wc_ecc_ctx_new(REQ_RESP_CLIENT, &rng);
if(ctx == NULL) {
    // error generating new ecEncCtx object
}
```

Example

```
WC_RNG rng;
ecEncCtx* ctx = wc_ecc_ctx_new(0, &rng);
```

C.22.2.41 function wc_ecc_ctx_free

```
void wc_ecc_ctx_free(
    ecEncCtx * ctx
)
```

This function frees the ecEncCtx object used for encrypting and decrypting messages.

Parameters:

- **ctx** pointer to the ecEncCtx object to free

See: wc_ecc_ctx_new

Return: none Returns.

Example

```
ecEncCtx* ctx;
WC_RNG rng;
wc_InitRng(&rng);
ctx = wc_ecc_ctx_new(REQ_RESP_CLIENT, &rng);
// do secure communication
...
wc_ecc_ctx_free(&ctx);
```

C.22.2.42 function wc_ecc_ctx_reset

```
int wc_ecc_ctx_reset(
    ecEncCtx * ctx,
    WC_RNG * rng
)
```

This function resets an ecEncCtx structure to avoid having to free and allocate a new context object.

Parameters:

- **ctx** pointer to the ecEncCtx object to reset
- **rng** pointer to an RNG object with which to generate a new salt
- **ctx** ECC encryption context
- **rng** Random number generator

See:

- wc_ecc_ctx_new
- wc_ecc_ctx_set_dev_id
- wc_ecc_ctx_new

Return:

- 0 Returned if the ecEncCtx structure is successfully reset
- BAD_FUNC_ARG Returned if either rng or ctx is NULL
- RNG_FAILURE_E Returned if there is an error generating a new salt for the ECC object
- 0 on success
- negative on error

Note: The device id set with wc_ecc_ctx_set_dev_id() (WOLF_CRYPTO_CB builds) is kept across the reset, like the heap hint.

Resets ECC encryption context.

Example

```
ecEncCtx* ctx;
WC_RNG rng;
wc_InitRng(&rng);
ctx = wc_ecc_ctx_new(REQ_RESP_CLIENT, &rng);
// do secure communication
...
wc_ecc_ctx_reset(&ctx, &rng);
// do more secure communication
```

Example

```
ecEncCtx* ctx;
WC_RNG rng;
int ret = wc_ecc_ctx_reset(ctx, &rng);
```

C.22.2.43 function wc_ecc_ctx_set_algo

```
int wc_ecc_ctx_set_algo(
    ecEncCtx * ctx,
    byte encAlgo,
    byte kdfAlgo,
    byte macAlgo
)
```

This function can optionally be called after wc_ecc_ctx_new. It sets the encryption, KDF, and MAC algorithms into an ecEncCtx object.

Parameters:

- **ctx** pointer to the ecEncCtx for which to set the info
- **encAlgo** encryption algorithm to use.
- **kdfAlgo** KDF algorithm to use.

- **macAlgo** MAC algorithm to use.

See: wc_ecc_ctx_new

Return:

- 0 Returned upon successfully setting the information for the ecEncCtx object.
- BAD_FUNC_ARG Returned if the given ecEncCtx object is NULL.

Example

```
ecEncCtx* ctx;
// initialize ctx
if(wc_ecc_ctx_set_algo(&ctx, ecAES_128_CTR, ecHKDF_SHA256, ecHMAC_SHA256))) {
    // error setting info
}
```

C.22.2.44 function wc_ecc_ctx_set_dev_id

```
int wc_ecc_ctx_set_dev_id(
    ecEncCtx * ctx,
    int devId
)
```

This function picks the device that ECIES operations using this context run on. Only available when WOLF_CRYPTO_CB is defined. A context starts at INVALID_DEVID, meaning software: ECIES does not copy the device from the private key, so this must be called for a crypto callback to be reached. The value is used both for the whole-operation ECIES callback and for the KDF, AES and HMAC steps of the software path. Passing a NULL context to wc_ecc_encrypt() or wc_ecc_decrypt() always means software. When WOLF_CRYPTO_CB_FIND is defined, an unset device id still goes through the registered finder, as it does for every other wolfCrypt operation. The setting is kept across wc_ecc_ctx_reset().

Parameters:

- **ctx** pointer to the ecEncCtx for which to set the device id
- **devId** device id to use, or INVALID_DEVID for software

See:

- wc_ecc_ctx_get_dev_id
- wc_ecc_ctx_new
- wc_ecc_ctx_reset

Return:

- 0 Returned upon successfully setting the device id.
- BAD_FUNC_ARG Returned if the given context is NULL.

Example

```
ecEncCtx* ctx = wc_ecc_ctx_new(REQ_RESP_CLIENT, &rng);
if (wc_ecc_ctx_set_dev_id(ctx, myDevId) != 0) {
    // error setting device id
}
```

C.22.2.45 function wc_ecc_ctx_get_dev_id

```
int wc_ecc_ctx_get_dev_id(
    ecEncCtx * ctx,
    int * devId
)
```

This function reads back the device id set with wc_ecc_ctx_set_dev_id(). Crypto callback code can use it to learn which device it was called for. Only available when WOLF_CRYPTO_CB is defined. A context that was never given a device reads back INVALID_DEVID.

Parameters:

- **ctx** pointer to the ecEncCtx to read the device id from
- **devId** pointer that receives the device id

See:

- wc_ecc_ctx_set_dev_id
- wc_ecc_ctx_new

Return:

- 0 Returned upon successfully reading the device id.
- BAD_FUNC_ARG Returned if the given context or output pointer is NULL.

Example

```
int devId;
if (wc_ecc_ctx_get_dev_id(ctx, &devId) != 0) {
    // error reading device id
}
```

C.22.2.46 function wc_ecc_ctx_get_own_salt

```
const byte * wc_ecc_ctx_get_own_salt(
    ecEncCtx * ctx
)
```

This function returns the salt of an ecEncCtx object. This function should only be called when the ecEncCtx's state is ecSRV_INIT or ecCLI_INIT.

Parameters:

- **ctx** pointer to the ecEncCtx object from which to get the salt
- **ctx** ECC encryption context

See:

- wc_ecc_ctx_new
- wc_ecc_ctx_set_peer_salt
- wc_ecc_ctx_set_kdf_salt
- wc_ecc_ctx_set_own_salt

Return:

- Success On success, returns the ecEncCtx salt
- NULL Returned if the ecEncCtx object is NULL, or the ecEncCtx's state is not ecSRV_INIT or ecCLI_INIT. In the latter two cases, this function also sets the ecEncCtx's state to ecSRV_BAD_STATE or ecCLI_BAD_STATE, respectively
- Salt pointer on success
- NULL on failure

Gets own salt from context.

Example

```
ecEncCtx* ctx;
WC_RNG rng;
const byte* salt;
```

```
wc_InitRng(&rng);
ctx = wc_ecc_ctx_new(REQ_RESP_CLIENT, &rng);
salt = wc_ecc_ctx_get_own_salt(&ctx);
if(salt == NULL) {
    // error getting salt
}
```

Example

```
ecEncCtx* ctx;
const byte* salt = wc_ecc_ctx_get_own_salt(ctx);
```

C.22.2.47 function wc_ecc_ctx_set_peer_salt

```
int wc_ecc_ctx_set_peer_salt(
    ecEncCtx * ctx,
    const byte * salt
)
```

This function sets the peer salt of an ecEncCtx object.

Parameters:

- **ctx** pointer to the ecEncCtx for which to set the salt
- **salt** pointer to the peer's salt

See:

- wc_ecc_ctx_get_own_salt
- wc_ecc_ctx_set_kdf_salt

Return:

- 0 Returned upon successfully setting the peer salt for the ecEncCtx object.
- BAD_FUNC_ARG Returned if the given ecEncCtx object is NULL or has an invalid protocol, or if the given salt is NULL
- BAD_ENC_STATE_E Returned if the ecEncCtx's state is ecSRV_SALT_GET or ecCLI_SALT_GET. In the latter two cases, this function also sets the ecEncCtx's state to ecSRV_BAD_STATE or ecCLI_BAD_STATE, respectively

Example

```
ecEncCtx* cliCtx, srvCtx;
WC_RNG rng;
const byte* cliSalt, srvSalt;
int ret;

wc_InitRng(&rng);
cliCtx = wc_ecc_ctx_new(REQ_RESP_CLIENT, &rng);
srvCtx = wc_ecc_ctx_new(REQ_RESP_SERVER, &rng);

cliSalt = wc_ecc_ctx_get_own_salt(&cliCtx);
srvSalt = wc_ecc_ctx_get_own_salt(&srvCtx);
ret = wc_ecc_ctx_set_peer_salt(&cliCtx, srvSalt);
```

C.22.2.48 function wc_ecc_ctx_set_kdf_salt

```
int wc_ecc_ctx_set_kdf_salt(
    ecEncCtx * ctx,
    const byte * salt,
```

```
    word32 sz
)
```

This function sets the salt pointer and length to use with KDF into the ecEncCtx object.

Parameters:

- **ctx** pointer to the ecEncCtx for which to set the salt
- **salt** pointer to salt buffer
- **sz** length salt in bytes

See:

- wc_ecc_ctx_get_own_salt
- wc_ecc_ctx_get_peer_salt

Return:

- 0 Returned upon successfully setting the salt for the ecEncCtx object.
- BAD_FUNC_ARG Returned if the given ecEncCtx object is NULL or if the given salt is NULL and length is not NULL.

Example

```
ecEncCtx* srvCtx;
WC_RNG rng;
byte cliSalt[] = { fixed salt data };
word32 cliSaltLen = (word32)sizeof(cliSalt);
int ret;

wc_InitRng(&rng);
cliCtx = wc_ecc_ctx_new(REQ_RESP_SERVER, &rng);

ret = wc_ecc_ctx_set_kdf_salt(&cliCtx, cliSalt, cliSaltLen);
```

C.22.2.49 function wc_ecc_ctx_set_info

```
int wc_ecc_ctx_set_info(
    ecEncCtx * ctx,
    const byte * info,
    int sz
)
```

This function can optionally be called before or after wc_ecc_ctx_set_peer_salt. It sets optional information for an ecEncCtx object.

Parameters:

- **ctx** pointer to the ecEncCtx for which to set the info
- **info** pointer to a buffer containing the info to set
- **sz** size of the info buffer

See: wc_ecc_ctx_new

Return:

- 0 Returned upon successfully setting the information for the ecEncCtx object.
- BAD_FUNC_ARG Returned if the given ecEncCtx object is NULL, the input info is NULL or it's size is invalid

Example

```
ecEncCtx* ctx;
byte info[] = { initialize with information };
// initialize ctx, get salt,
if(wc_ecc_ctx_set_info(&ctx, info, sizeof(info))) {
    // error setting info
}
```

C.22.2.50 function wc_ecc_encrypt

```
int wc_ecc_encrypt(
    ecc_key * privKey,
    ecc_key * pubKey,
    const byte * msg,
    word32 msgSz,
    byte * out,
    word32 * outSz,
    ecEncCtx * ctx
)
```

This function encrypts the given input message from msg to out. This function takes an optional ctx object as parameter. When supplied, encryption proceeds based on the ecEncCtx's encAlgo, kdfAlgo, and macAlgo. If ctx is not supplied, processing completes with the default algorithms, ecAES_128_CBC, ecHKDF_SHA256 and ecHMAC_SHA256. This function requires that the messages are padded according to the encryption type specified by ctx.

Parameters:

- **privKey** pointer to the ecc_key object containing the private key to use for encryption
- **pubKey** pointer to the ecc_key object containing the public key of the peer with whom one wishes to communicate
- **msg** pointer to the buffer holding the message to encrypt
- **msgSz** size of the buffer to encrypt
- **out** pointer to the buffer in which to store the encrypted ciphertext
- **outSz** pointer to a word32 object containing the available size in the out buffer. Upon successfully encrypting the message, holds the number of bytes written to the output buffer
- **ctx** Optional: pointer to an ecEncCtx object specifying different encryption algorithms to use

See:

- wc_ecc_encrypt_ex
- wc_ecc_decrypt
- wc_ecc_ctx_set_dev_id

Return:

- 0 Returned upon successfully encrypting the input message
- BAD_FUNC_ARG Returned if privKey, pubKey, msg, msgSz, out, or outSz are NULL, or the ctx object specifies an unsupported encryption type
- BAD_ENC_STATE_E Returned if the ctx object given is in a state that is not appropriate for encryption
- BUFFER_E Returned if the supplied output buffer is too small to store the encrypted ciphertext
- MEMORY_E Returned if there is an error allocating memory for the shared secret key

Note:

- Selecting an AES-GCM DEM algorithm (ecAES_128_GCM, ecAES_256_GCM) in the default IV mode requires the WOLFSSL_ECIES_STATIC_GCM_NONCE build macro; otherwise this function returns NOT_COMPILED_IN. See wc_ecc_encrypt_ex for the full rationale.

- The device this runs on comes from the context (wc_ecc_ctx_set_dev_id), not from privKey->devId. A NULL context, or one that was never given a device, runs in software.

Example

```
byte msg[] = { initialize with msg to encrypt. Ensure padded to block size };
byte out[sizeof(msg)];
word32 outSz = sizeof(out);
int ret;
ecc_key cli, serv;
// initialize cli with private key
// initialize serv with received public key

ecEncCtx* cliCtx, servCtx;
// initialize cliCtx and servCtx
// exchange salts
ret = wc_ecc_encrypt(&cli, &serv, msg, sizeof(msg), out, &outSz, cliCtx);
if(ret != 0) {
    // error encrypting message
}
```

C.22.2.51 function wc_ecc_encrypt_ex

```
int wc_ecc_encrypt_ex(
    ecc_key * privKey,
    ecc_key * pubKey,
    const byte * msg,
    word32 msgSz,
    byte * out,
    word32 * outSz,
    ecEncCtx * ctx,
    int compressed
)
```

This function encrypts the given input message from msg to out. This function takes an optional ctx object as parameter. When supplied, encryption proceeds based on the ecEncCtx's encAlgo, kdfAlgo, and macAlgo. If ctx is not supplied, processing completes with the default algorithms, ecAES_128_CBC, ecHKDF_SHA256 and ecHMAC_SHA256. This function requires that the messages are padded according to the encryption type specified by ctx.

Parameters:

- **privKey** pointer to the ecc_key object containing the private key to use for encryption
- **pubKey** pointer to the ecc_key object containing the public key of the peer with whom one wishes to communicate
- **msg** pointer to the buffer holding the message to encrypt
- **msgSz** size of the buffer to encrypt
- **out** pointer to the buffer in which to store the encrypted ciphertext
- **outSz** pointer to a word32 object containing the available size in the out buffer. Upon successfully encrypting the message, holds the number of bytes written to the output buffer
- **ctx** Optional: pointer to an ecEncCtx object specifying different encryption algorithms to use
- **compressed** Public key field is to be output in compressed format.

See:

- wc_ecc_encrypt
- wc_ecc_decrypt
- wc_ecc_ctx_set_dev_id

Return:

- 0 Returned upon successfully encrypting the input message
- BAD_FUNC_ARG Returned if privKey, pubKey, msg, msgSz, out, or outSz are NULL, or the ctx object specifies an unsupported encryption type
- BAD_ENC_STATE_E Returned if the ctx object given is in a state that is not appropriate for encryption
- BUFFER_E Returned if the supplied output buffer is too small to store the encrypted ciphertext
- MEMORY_E Returned if there is an error allocating memory for the shared secret key
- NOT_COMPILED_IN Returned if an AES-GCM DEM algorithm (ecAES_128_GCM or ecAES_256_GCM) is requested in the default IV mode without the WOLFSSL_ECIES_STATIC_GCM_NONCE build macro defined

Note:

- The AES-GCM DEM algorithms (ecAES_128_GCM, ecAES_256_GCM) in the default IV mode use a fixed all-zero GCM nonce. That is safe only because ECIES derives a fresh symmetric key from a fresh ephemeral key on every encryption, so the (key, nonce) pair never repeats; reusing the ephemeral key is catastrophic for AES-GCM. For that reason the fixed-nonce GCM DEM is off by default and must be enabled with the WOLFSSL_ECIES_STATIC_GCM_NONCE build macro, otherwise this function returns NOT_COMPILED_IN for a GCM algorithm. The macro is not needed with WOLFSSL_ECIES_GEN_IV (random per-message nonce) or WOLFSSL_ECIES_OLD (nonce derived from the KDF output).
- The device this runs on comes from the context (wc_ecc_ctx_set_dev_id), not from privKey->devId. A NULL context, or one that was never given a device, runs in software.

Example

```
byte msg[] = { initialize with msg to encrypt. Ensure padded to block size };
byte out[sizeof(msg)];
word32 outSz = sizeof(out);
int ret;
ecc_key cli, serv;
// initialize cli with private key
// initialize serv with received public key

ecEncCtx* cliCtx, servCtx;
// initialize cliCtx and servCtx
// exchange salts
ret = wc_ecc_encrypt_ex(&cli, &serv, msg, sizeof(msg), out, &outSz, cliCtx,
    1);
if(ret != 0) {
    // error encrypting message
}
```

C.22.2.52 function wc_ecc_decrypt

```
int wc_ecc_decrypt(
    ecc_key * privKey,
    ecc_key * pubKey,
    const byte * msg,
    word32 msgSz,
    byte * out,
    word32 * outSz,
    ecEncCtx * ctx
)
```

This function decrypts the ciphertext from msg to out. This function takes an optional ctx object as parameter. When supplied, encryption proceeds based on the ecEncCtx's encAlgo, kdfAlgo, and macAlgo. If ctx is not supplied, processing completes with the default algorithms, ecAES_128_CBC, ecHKDF_SHA256 and ecHMAC_SHA256. This function requires that the messages are padded according to the encryption type specified by ctx.

Parameters:

- **privKey** pointer to the ecc_key object containing the private key to use for decryption
- **pubKey** only used when built with WOLFSSL_ECIES_OLD: pointer to the ecc_key object containing the public key of the peer with whom one wishes to communicate. In the default message format the sender's ephemeral public key is read from the start of msg instead and pubKey is ignored (it may be NULL and is left unmodified)
- **msg** pointer to the buffer holding the ciphertext to decrypt
- **msgSz** size of the buffer to decrypt
- **out** pointer to the buffer in which to store the decrypted plaintext
- **outSz** pointer to a word32 object containing the available size in the out buffer. Upon successfully decrypting the ciphertext, holds the number of bytes written to the output buffer
- **ctx** Optional: pointer to an ecEncCtx object specifying different decryption algorithms to use

See:

- wc_ecc_encrypt
- wc_ecc_encrypt_ex
- wc_ecc_ctx_set_dev_id

Return:

- 0 Returned upon successfully decrypting the input message
- BAD_FUNC_ARG Returned if privKey, msg, msgSz, out, or outSz are NULL (or pubKey is NULL when built with WOLFSSL_ECIES_OLD), or the ctx object specifies an unsupported encryption type
- BAD_ENC_STATE_E Returned if the ctx object given is in a state that is not appropriate for decryption
- BUFFER_E Returned if the supplied output buffer is too small to store the decrypted plaintext
- MEMORY_E Returned if there is an error allocating memory for the shared secret key
- NOT_COMPILED_IN Returned if an AES-GCM DEM algorithm (ecAES_128_GCM or ecAES_256_GCM) is requested in the default IV mode without the WOLFSSL_ECIES_STATIC_GCM_NONCE build macro defined

Note:

- Selecting an AES-GCM DEM algorithm (ecAES_128_GCM, ecAES_256_GCM) in the default IV mode requires the WOLFSSL_ECIES_STATIC_GCM_NONCE build macro; otherwise this function returns NOT_COMPILED_IN. See wc_ecc_encrypt_ex for the full rationale.
- The device this runs on comes from the context (wc_ecc_ctx_set_dev_id), not from privKey->devId. A NULL context, or one that was never given a device, runs in software.

Example

```
byte cipher[] = { initialize with
ciphertext to decrypt. Ensure padded to block size };
byte plain[sizeof(cipher)];
word32 plainSz = sizeof(plain);
int ret;
ecc_key cli, serv;
// initialize cli with private key
// initialize serv with received public key
ecEncCtx* cliCtx, servCtx;
// initialize cliCtx and servCtx
```

```
// exchange salts
ret = wc_ecc_decrypt(&cli, &serv, cipher, sizeof(cipher),
plain, &plainSz, cliCtx);

if(ret != 0) {
    // error decrypting message
}
```

C.22.2.53 function wc_ecc_set_nonblock

```
int wc_ecc_set_nonblock(
    ecc_key * key,
    ecc_nb_ctx_t * ctx
)
```

Enable ECC support for non-blocking operations. Supported for Single Precision (SP) math with the following build options: WOLFSSL_SP_NONBLOCK WOLFSSL_SP_SMALL WOLFSSL_SP_NO_MALLOC WC_ECC_NONBLOCK.

Parameters:

- **key** pointer to the ecc_key object
- **ctx** pointer to ecc nb_ctx_t structure with stack data cache for SP

Return: 0 Returned upon successfully setting the callback context the input message

Example

```
int ret;
ecc_key ecc;
ecc_nb_ctx_t nb_ctx;

ret = wc_ecc_init(&ecc);
if (ret == 0) {
    ret = wc_ecc_set_nonblock(&ecc, &nb_ctx);
    if (ret == 0) {
        do {
            ret = wc_ecc_verify_hash_ex(
                &r, &s,       // r/s as mp_int
                hash, hashSz, // computed hash digest
                &verify_res,  // verification result 1=success
                &key
            );

            // TODO: Real-time work can be called here
        } while (ret == FP_WOULDBLOCK);
    }
    wc_ecc_free(&key);
}
```

C.22.2.54 function wc_ecc_set_curve

```
int wc_ecc_set_curve(
    ecc_key * key,
    int keysize,
    int curve_id
)
```

Compare a curve which has larger key than specified size or the curve matched curve ID, set a curve with smaller key size to the key.

Parameters:

- **keysize** Key size in bytes
- **curve_id** Curve ID

```
 int ret;
ecc_key ecc;

ret = wc_ecc_init(&ecc);
if (ret != 0)
    return ret;
    ret = wc_ecc_set_curve(&ecc, 32, ECC_SECP256R1));
    if (ret != 0)
        return ret;
```

Return: 0 Returned upon successfully setting the key

C.22.2.55 function wc_ecc_key_get_priv

```
mp_int * wc_ecc_key_get_priv(
    ecc_key * key
)
```

Gets private key mp_int from ECC key.

Parameters:

- **key** ECC key structure

See: wc_ecc_init

Return:

- mp_int pointer on success
- NULL on failure

Example

```
ecc_key key;
mp_int* priv = wc_ecc_key_get_priv(&key);
```

C.22.2.56 function wc_ecc_get_sets_count

```
size_t wc_ecc_get_sets_count(
    void
)
```

Returns number of supported ECC curve sets.

See: wc_ecc_get_curve_params

Return: Number of curve sets

Example

```
size_t count = wc_ecc_get_sets_count();
```

C.22.2.57 function wc_ecc_get_name

```
const char * wc_ecc_get_name(
    int curve_id
)
```

Gets curve name from curve ID.

Parameters:

- **curve_id** Curve identifier

See: wc_ecc_get_curve_id

Return:

- Curve name string on success
- NULL on failure

Example

```
const char* name = wc_ecc_get_name(ECC_SECP256R1);
```

C.22.2.58 function wc_ecc_make_key_ex2

```
int wc_ecc_make_key_ex2(
    WC_RNG * rng,
    int keysize,
    ecc_key * key,
    int curve_id,
    int flags
)
```

Makes ECC key with extended options.

Parameters:

- **rng** Random number generator
- **keysize** Key size in bytes
- **key** ECC key structure
- **curve_id** Curve identifier
- **flags** Additional flags

See: wc_ecc_make_key_ex

Return:

- 0 on success
- negative on error

Example

```
WC_RNG rng;
ecc_key key;
int ret = wc_ecc_make_key_ex2(&rng, 32, &key,
                              ECC_SECP256R1, 0);
```

C.22.2.59 function wc_ecc_is_point

```
int wc_ecc_is_point(
    ecc_point * ecp,
    mp_int * a,
    mp_int * b,
```

```
    mp_int * prime
)
```

Checks if point is on curve.

Parameters:

- **ecp** ECC point
- **a** Curve parameter a
- **b** Curve parameter b
- **prime** Curve prime

See: wc_ecc_point_is_on_curve

Return:

- 1 if point is on curve
- 0 if not on curve
- negative on error

Example

```
ecc_point* point;
mp_int a, b, prime;
int ret = wc_ecc_is_point(point, &a, &b, &prime);
```

C.22.2.60 function wc_ecc_get_generator

```
int wc_ecc_get_generator(
    ecc_point * ecp,
    int curve_idx
)
```

Gets generator point for curve.

Parameters:

- **ecp** ECC point to store generator
- **curve_idx** Curve index

See: wc_ecc_get_curve_params

Return:

- 0 on success
- negative on error

Example

```
ecc_point* gen = wc_ecc_new_point();
int ret = wc_ecc_get_generator(gen, 0);
```

C.22.2.61 function wc_ecc_set_deterministic

```
int wc_ecc_set_deterministic(
    ecc_key * key,
    byte flag
)
```

Sets deterministic signing mode.

Parameters:

- **key** ECC key

- **flag** Enable/disable flag

See: wc_ecc_set_deterministic_ex

Return:

- 0 on success
- negative on error

Example

```
ecc_key key;
int ret = wc_ecc_set_deterministic(&key, 1);
```

C.22.2.62 function wc_ecc_set_deterministic_ex

```
int wc_ecc_set_deterministic_ex(
    ecc_key * key,
    byte flag,
    enum wc_HashType hashType
)
```

Sets deterministic signing with hash type.

Parameters:

- **key** ECC key
- **flag** Enable/disable flag
- **hashType** Hash algorithm type

See: wc_ecc_set_deterministic

Return:

- 0 on success
- negative on error

Example

```
ecc_key key;
int ret = wc_ecc_set_deterministic_ex(&key, 1, WC_HASH_TYPE_SHA256);
```

C.22.2.63 function wc_ecc_gen_deterministic_k

```
int wc_ecc_gen_deterministic_k(
    const byte * hash,
    word32 hashSz,
    enum wc_HashType hashType,
    mp_int * priv,
    mp_int * k,
    mp_int * order,
    void * heap
)
```

Generates deterministic k value for signing.

Parameters:

- **hash** Hash value
- **hashSz** Hash size
- **hashType** Hash algorithm type
- **priv** Private key

- **k** Output k value
- **order** Curve order
- **heap** Heap hint

See: wc_ecc_sign_set_k

Return:

- 0 on success
- negative on error

Example

```
byte hash[32];
mp_int priv, k, order;
int ret = wc_ecc_gen_deterministic_k(hash, 32,
                                     WC_HASH_TYPE_SHA256,
                                     &priv, &k, &order, NULL);
```

C.22.2.64 function wc_ecc_sign_set_k

```
int wc_ecc_sign_set_k(
    const byte * k,
    word32 klen,
    ecc_key * key
)
```

Sets k value for signing.

Parameters:

- **k** K value buffer
- **klen** K value length
- **key** ECC key

See: wc_ecc_gen_deterministic_k

Return:

- 0 on success
- negative on error

Example

```
byte k[32];
ecc_key key;
int ret = wc_ecc_sign_set_k(k, sizeof(k), &key);
```

C.22.2.65 function wc_ecc_init_id

```
int wc_ecc_init_id(
    ecc_key * key,
    unsigned char * id,
    int len,
    void * heap,
    int devId
)
```

Initializes ECC key with ID.

Parameters:

- **key** ECC key
- **id** ID buffer
- **len** ID length
- **heap** Heap hint
- **devId** Device ID

See: wc_ecc_init_label

Return:

- 0 on success
- negative on error

Note: This API is only available when WOLF_PRIVATE_KEY_ID is defined, which is set for PKCS11 support.

Example

```
ecc_key key;
unsigned char id[] = "mykey";
int ret = wc_ecc_init_id(&key, id, sizeof(id), NULL,
                         INVALID_DEVID);
```

C.22.2.66 function wc_ecc_init_label

```
int wc_ecc_init_label(
    ecc_key * key,
    const char * label,
    void * heap,
    int devId
)
```

Initializes ECC key with label.

Parameters:

- **key** ECC key
- **label** Label string
- **heap** Heap hint
- **devId** Device ID

See: wc_ecc_init_id

Return:

- 0 on success
- negative on error

Note: This API is only available when WOLF_PRIVATE_KEY_ID is defined, which is set for PKCS11 support.

Example

```
ecc_key key;
int ret = wc_ecc_init_label(&key, "mykey", NULL,
                            INVALID_DEVID);
```

C.22.2.67 function wc_ecc_set_flags

```
int wc_ecc_set_flags(
    ecc_key * key,
```

```
    word32 flags
)
```

Sets flags on ECC key.

Parameters:

- **key** ECC key
- **flags** Flags to set

See: wc_ecc_init

Return:

- 0 on success
- negative on error

Example

```
ecc_key key;
int ret = wc_ecc_set_flags(&key, WC_ECC_FLAG_COFACTOR);
```

C.22.2.68 function wc_ecc_fp_init

```
void wc_ecc_fp_init(
    void
)
```

Initializes fixed-point cache.

See: wc_ecc_init

Return: none No returns

Example

```
wc_ecc_fp_init();
```

C.22.2.69 function wc_ecc_set_rng

```
int wc_ecc_set_rng(
    ecc_key * key,
    WC_RNG * rng
)
```

Sets RNG for ECC key.

Parameters:

- **key** ECC key
- **rng** Random number generator

See: wc_ecc_make_key

Return:

- 0 on success
- negative on error

Example

```
ecc_key key;
WC_RNG rng;
int ret = wc_ecc_set_rng(&key, &rng);
```

C.22.2.70 function wc_ecc_get_curve_idx

```
int wc_ecc_get_curve_idx(
    int curve_id
)
```

Gets curve index from curve ID.

Parameters:

- **curve_id** Curve identifier

See: wc_ecc_get_curve_id

Return:

- Curve index on success
- negative on error

Example

```
int idx = wc_ecc_get_curve_idx(ECC_SECP256R1);
```

C.22.2.71 function wc_ecc_get_curve_id

```
int wc_ecc_get_curve_id(
    int curve_idx
)
```

Gets curve ID from curve index.

Parameters:

- **curve_idx** Curve index

See: wc_ecc_get_curve_idx

Return:

- Curve ID on success
- negative on error

Example

```
int id = wc_ecc_get_curve_id(0);
```

C.22.2.72 function wc_ecc_get_curve_size_from_id

```
int wc_ecc_get_curve_size_from_id(
    int curve_id
)
```

Gets curve size from curve ID.

Parameters:

- **curve_id** Curve identifier

See: wc_ecc_get_curve_id

Return:

- Key size in bytes on success
- negative on error

Example

```
int size = wc_ecc_get_curve_size_from_id(ECC_SECP256R1);
```

C.22.2.73 function wc_ecc_get_curve_idx_from_name

```
int wc_ecc_get_curve_idx_from_name(
    const char * curveName
)
```

Gets curve index from curve name.

Parameters:

- **curveName** Curve name string

See: wc_ecc_get_name

Return:

- Curve index on success
- negative on error

Example

```
int idx = wc_ecc_get_curve_idx_from_name("SECP256R1");
```

C.22.2.74 function wc_ecc_get_curve_size_from_name

```
int wc_ecc_get_curve_size_from_name(
    const char * curveName
)
```

Gets curve size from curve name.

Parameters:

- **curveName** Curve name string

See: wc_ecc_get_curve_idx_from_name

Return:

- Key size in bytes on success
- negative on error

Example

```
int size = wc_ecc_get_curve_size_from_name("SECP256R1");
```

C.22.2.75 function wc_ecc_get_curve_id_from_name

```
int wc_ecc_get_curve_id_from_name(
    const char * curveName
)
```

Gets curve ID from curve name.

Parameters:

- **curveName** Curve name string

See: wc_ecc_get_name

Return:

- Curve ID on success

- negative on error

Example

```
int id = wc_ecc_get_curve_id_from_name("SECP256R1");
```

C.22.2.76 function wc_ecc_get_curve_id_from_params

```
int wc_ecc_get_curve_id_from_params(
    int fieldSize,
    const byte * prime,
    word32 primeSz,
    const byte * Af,
    word32 AfSz,
    const byte * Bf,
    word32 BfSz,
    const byte * order,
    word32 orderSz,
    const byte * Gx,
    word32 GxSz,
    const byte * Gy,
    word32 GySz,
    int cofactor
)
```

Gets curve ID from curve parameters.

Parameters:

- **fieldSize** Field size
- **prime** Prime modulus
- **primeSz** Prime size
- **Af** Curve parameter A
- **AfSz** A size
- **Bf** Curve parameter B
- **BfSz** B size
- **order** Curve order
- **orderSz** Order size
- **Gx** Generator X coordinate
- **GxSz** Gx size
- **Gy** Generator Y coordinate
- **GySz** Gy size
- **cofactor** Curve cofactor

See: wc_ecc_get_curve_params

Return:

- Curve ID on success
- negative on error

Example

```
int id = wc_ecc_get_curve_id_from_params(256, prime, 32,
                                         Af, 32, Bf, 32,
                                         order, 32, Gx, 32,
                                         Gy, 32, 1);
```

C.22.2.77 function wc_ecc_get_curve_id_from_dp_params

```
int wc_ecc_get_curve_id_from_dp_params(
    const ecc_set_type * dp
)
```

Gets curve ID from domain parameters.

Parameters:

- **dp** Domain parameters

See: wc_ecc_get_curve_params

Return:

- Curve ID on success
- negative on error

Example

```
const ecc_set_type* dp;
int id = wc_ecc_get_curve_id_from_dp_params(dp);
```

C.22.2.78 function wc_ecc_get_curve_id_from_oid

```
int wc_ecc_get_curve_id_from_oid(
    const byte * oid,
    word32 len
)
```

Gets curve ID from OID.

Parameters:

- **oid** OID buffer
- **len** OID length

See: wc_ecc_get_oid

Return:

- Curve ID on success
- negative on error

Example

```
byte oid[] = {0x2A, 0x86, 0x48, 0xCE, 0x3D, 0x03, 0x01, 0x07};
int id = wc_ecc_get_curve_id_from_oid(oid, sizeof(oid));
```

C.22.2.79 function wc_ecc_get_curve_params

```
const ecc_set_type * wc_ecc_get_curve_params(
    int curve_idx
)
```

Gets curve parameters from curve index.

Parameters:

- **curve_idx** Curve index

See: wc_ecc_get_curve_idx

Return:

- ecc_set_type pointer on success
- NULL on failure

Example

```
const ecc_set_type* params = wc_ecc_get_curve_params(0);
```

C.22.2.80 function wc_ecc_new_point_h

```
ecc_point * wc_ecc_new_point_h(
    void * h
)
```

Allocates new ECC point with heap hint.

Parameters:

- **h** Heap hint

See: wc_ecc_del_point_h

Return:

- ecc_point pointer on success
- NULL on failure

Example

```
ecc_point* point = wc_ecc_new_point_h(NULL);
if (point != NULL) {
    // use point
    wc_ecc_del_point_h(point, NULL);
}
```

C.22.2.81 function wc_ecc_del_point_h

```
void wc_ecc_del_point_h(
    ecc_point * p,
    void * h
)
```

Frees ECC point with heap hint.

Parameters:

- **p** ECC point to free
- **h** Heap hint

See: wc_ecc_new_point_h

Return: none No returns

Example

```
ecc_point* point = wc_ecc_new_point_h(NULL);
// use point
wc_ecc_del_point_h(point, NULL);
```

C.22.2.82 function wc_ecc_forcezero_point

```
void wc_ecc_forcezero_point(
    ecc_point * p
)
```

Securely zeros ECC point.

Parameters:

- **p** ECC point to zero

See: wc_ecc_del_point

Return: none No returns

Example

```
ecc_point* point;
wc_ecc_forcezero_point(point);
```

C.22.2.83 function wc_ecc_point_is_on_curve

```
int wc_ecc_point_is_on_curve(
    ecc_point * p,
    int curve_idx
)
```

Checks if point is on curve.

Parameters:

- **p** ECC point
- **curve_idx** Curve index

See: wc_ecc_is_point

Return:

- 1 if on curve
- 0 if not on curve
- negative on error

Example

```
ecc_point* point;
int ret = wc_ecc_point_is_on_curve(point, 0);
```

C.22.2.84 function wc_ecc_import_x963_ex

```
int wc_ecc_import_x963_ex(
    const byte * in,
    word32 inLen,
    ecc_key * key,
    int curve_id
)
```

Imports X9.63 format with curve ID.

Parameters:

- **in** Input buffer
- **inLen** Input length
- **key** ECC key
- **curve_id** Curve identifier

See: wc_ecc_import_x963

Return:

- 0 on success
- negative on error

Example

```
byte x963[65];
ecc_key key;
int ret = wc_ecc_import_x963_ex(x963, sizeof(x963), &key,
                                ECC_SECP256R1);
```

C.22.2.85 function wc_ecc_import_private_key_ex

```
int wc_ecc_import_private_key_ex(
    const byte * priv,
    word32 privSz,
    const byte * pub,
    word32 pubSz,
    ecc_key * key,
    int curve_id
)
```

Imports private key with curve ID.

Parameters:

- **priv** Private key buffer
- **privSz** Private key size
- **pub** Public key buffer
- **pubSz** Public key size
- **key** ECC key
- **curve_id** Curve identifier

See: wc_ecc_import_private_key

Return:

- 0 on success
- negative on error

Example

```
byte priv[32], pub[65];
ecc_key key;
int ret = wc_ecc_import_private_key_ex(priv, 32, pub, 65,
                                       &key, ECC_SECP256R1);
```

C.22.2.86 function wc_ecc_rs_raw_to_sig

```
int wc_ecc_rs_raw_to_sig(
    const byte * r,
    word32 rSz,
    const byte * s,
    word32 sSz,
    byte * out,
    word32 * outlen
)
```

Converts raw r,s to signature.

Parameters:

- **r** R value buffer
- **rSz** R value size
- **s** S value buffer
- **sSz** S value size
- **out** Output signature buffer
- **outlen** Output signature length

See: wc_ecc_sig_to_rs

Return:

- 0 on success
- negative on error

Example

```
byte r[32], s[32], sig[72];
word32 sigLen = sizeof(sig);
int ret = wc_ecc_rs_raw_to_sig(r, 32, s, 32, sig, &sigLen);
```

C.22.2.87 function wc_ecc_sig_to_rs

```
int wc_ecc_sig_to_rs(
    const byte * sig,
    word32 sigLen,
    byte * r,
    word32 * rLen,
    byte * s,
    word32 * sLen
)
```

Converts signature to raw r,s.

Parameters:

- **sig** Signature buffer
- **sigLen** Signature length
- **r** R value buffer
- **rLen** R value length
- **s** S value buffer
- **sLen** S value length

See: wc_ecc_rs_raw_to_sig

Return:

- 0 on success
- negative on error

Example

```
byte sig[72], r[32], s[32];
word32 rLen = 32, sLen = 32;
int ret = wc_ecc_sig_to_rs(sig, 72, r, &rLen, s, &sLen);
```

C.22.2.88 function wc_ecc_import_raw_ex

```
int wc_ecc_import_raw_ex(
    ecc_key * key,
    const char * qx,
    const char * qy,
```

```
    const char * d,
    int curve_id
)
```

Imports raw key with curve ID.

Parameters:

- **key** ECC key
- **qx** X coordinate string
- **qy** Y coordinate string
- **d** Private key string
- **curve_id** Curve identifier

See: wc_ecc_import_raw

Return:

- 0 on success
- negative on error

Note: This function does not check that the imported public point lies on the curve. Define WOLFSSL_VALIDATE_ECC_IMPORT to validate the point on import, or call wc_ecc_check_key before the key is used.

Example

```
ecc_key key;
int ret = wc_ecc_import_raw_ex(&key, qxStr, qyStr, dStr,
                               ECC_SECP256R1);
```

C.22.2.89 function wc_ecc_import_unsigned

```
int wc_ecc_import_unsigned(
    ecc_key * key,
    const byte * qx,
    const byte * qy,
    const byte * d,
    int curve_id
)
```

Imports unsigned key with curve ID.

Parameters:

- **key** ECC key
- **qx** X coordinate buffer
- **qy** Y coordinate buffer
- **d** Private key buffer
- **curve_id** Curve identifier

See: wc_ecc_import_raw_ex

Return:

- 0 on success
- negative on error

Note: This function does not check that the imported public point lies on the curve. Define WOLFSSL_VALIDATE_ECC_IMPORT to validate the point on import, or call wc_ecc_check_key before the key is used.

Example

```
ecc_key key;
byte qx[32], qy[32], d[32];
int ret = wc_ecc_import_unsigned(&key, qx, qy, d,
                                 ECC_SECP256R1);
```

C.22.2.90 function wc_ecc_export_ex

```
int wc_ecc_export_ex(
    ecc_key * key,
    byte * qx,
    word32 * qxLen,
    byte * qy,
    word32 * qyLen,
    byte * d,
    word32 * dLen,
    int encType
)
```

Exports key with encoding type.

Parameters:

- **key** ECC key
- **qx** X coordinate buffer
- **qxLen** X coordinate length
- **qy** Y coordinate buffer
- **qyLen** Y coordinate length
- **d** Private key buffer
- **dLen** Private key length
- **encType** Encoding type

See: wc_ecc_export_public_raw

Return:

- 0 on success
- negative on error

Example

```
ecc_key key;
byte qx[32], qy[32], d[32];
word32 qxLen = 32, qyLen = 32, dLen = 32;
int ret = wc_ecc_export_ex(&key, qx, &qxLen, qy, &qyLen,
                           d, &dLen, 0);
```

C.22.2.91 function wc_ecc_export_public_raw

```
int wc_ecc_export_public_raw(
    ecc_key * key,
    byte * qx,
    word32 * qxLen,
    byte * qy,
    word32 * qyLen
)
```

Exports public key in raw format.

Parameters:

- **key** ECC key
- **qx** X coordinate buffer
- **qxLen** X coordinate length
- **qy** Y coordinate buffer
- **qyLen** Y coordinate length

See: wc_ecc_export_private_raw

Return:

- 0 on success
- negative on error

Example

```
ecc_key key;
byte qx[32], qy[32];
word32 qxLen = 32, qyLen = 32;
int ret = wc_ecc_export_public_raw(&key, qx, &qxLen, qy,
                                   &qyLen);
```

C.22.2.92 function wc_ecc_export_private_raw

```
int wc_ecc_export_private_raw(
    ecc_key * key,
    byte * qx,
    word32 * qxLen,
    byte * qy,
    word32 * qyLen,
    byte * d,
    word32 * dLen
)
```

Exports private key in raw format.

Parameters:

- **key** ECC key
- **qx** X coordinate buffer
- **qxLen** X coordinate length
- **qy** Y coordinate buffer
- **qyLen** Y coordinate length
- **d** Private key buffer
- **dLen** Private key length

See: wc_ecc_export_public_raw

Return:

- 0 on success
- negative on error

Example

```
ecc_key key;
byte qx[32], qy[32], d[32];
word32 qxLen = 32, qyLen = 32, dLen = 32;
int ret = wc_ecc_export_private_raw(&key, qx, &qxLen, qy,
                                    &qyLen, d, &dLen);
```

C.22.2.93 function wc_ecc_export_point_der_ex

```
int wc_ecc_export_point_der_ex(
    const int curve_idx,
    ecc_point * point,
    byte * out,
    word32 * outLen,
    int compressed
)
```

Exports point in DER format with compression.

Parameters:

- **curve_idx** Curve index
- **point** ECC point
- **out** Output buffer
- **outLen** Output length
- **compressed** Compression flag

See: wc_ecc_export_point_der

Return:

- Size on success
- negative on error

Example

```
ecc_point* point;
byte out[65];
word32 outLen = sizeof(out);
int ret = wc_ecc_export_point_der_ex(0, point, out, &outLen,
                                     0);
```

C.22.2.94 function wc_ecc_import_point_der_ex

```
int wc_ecc_import_point_der_ex(
    const byte * in,
    word32 inLen,
    const int curve_idx,
    ecc_point * point,
    int shortKeySize
)
```

Imports point from DER format.

Parameters:

- **in** Input buffer
- **inLen** Input length
- **curve_idx** Curve index
- **point** ECC point
- **shortKeySize** Short key size flag

See: wc_ecc_import_point_der

Return:

- 0 on success
- negative on error

Example

```
byte der[65];
ecc_point* point = wc_ecc_new_point();
int ret = wc_ecc_import_point_der_ex(der, sizeof(der), 0,
                                     point, 0);
```

C.22.2.95 function wc_ecc_get_oid

```
int wc_ecc_get_oid(
    word32 oidSum,
    const byte ** oid,
    word32 * oidSz
)
```

Gets OID for curve.

Parameters:

- **oidSum** OID sum
- **oid** OID buffer pointer
- **oidSz** OID size pointer

See: wc_ecc_get_curve_id_from_oid

Return:

- 0 on success
- negative on error

Example

```
const byte* oid;
word32 oidSz;
int ret = wc_ecc_get_oid(0x2A8648CE3D030107, &oid, &oidSz);
```

C.22.2.96 function wc_ecc_set_custom_curve

```
int wc_ecc_set_custom_curve(
    ecc_key * key,
    const ecc_set_type * dp
)
```

Sets custom curve parameters.

Parameters:

- **key** ECC key
- **dp** Domain parameters

See: wc_ecc_get_curve_params

Return:

- 0 on success
- negative on error

Example

```
ecc_key key;
ecc_set_type dp;
int ret = wc_ecc_set_custom_curve(&key, &dp);
```

C.22.2.97 function wc_ecc_ctx_new_ex

```
ecEncCtx * wc_ecc_ctx_new_ex(
    int flags,
    WC_RNG * rng,
    void * heap
)
```

Creates new ECC encryption context with heap.

Parameters:

- **flags** Context flags
- **rng** Random number generator
- **heap** Heap hint

See: wc_ecc_ctx_new

Return:

- ecEncCtx pointer on success
- NULL on failure

Example

```
WC_RNG rng;
ecEncCtx* ctx = wc_ecc_ctx_new_ex(0, &rng, NULL);
```

C.22.2.98 function wc_ecc_ctx_set_own_salt

```
int wc_ecc_ctx_set_own_salt(
    ecEncCtx * ctx,
    const byte * salt,
    word32 sz
)
```

Sets own salt in context.

Parameters:

- **ctx** ECC encryption context
- **salt** Salt buffer
- **sz** Salt size

See: wc_ecc_ctx_get_own_salt

Return:

- 0 on success
- negative on error

Example

```
ecEncCtx* ctx;
byte salt[16];
int ret = wc_ecc_ctx_set_own_salt(ctx, salt, sizeof(salt));
```

C.22.2.99 function wc_X963_KDF

```
int wc_X963_KDF(
    enum wc_HashType type,
    const byte * secret,
    word32 secretSz,
```

```
    const byte * sinfo,
    word32 sinfoSz,
    byte * out,
    word32 outSz
)
```

X9.63 Key Derivation Function.

Parameters:

- **type** Hash type
- **secret** Shared secret
- **secretSz** Secret size
- **sinfo** Shared info
- **sinfoSz** Shared info size
- **out** Output buffer
- **outSz** Output size

See: wc_ecc_shared_secret

Return:

- 0 on success
- negative on error

Example

```
byte secret[32], sinfo[10], out[32];
int ret = wc_X963_KDF(WC_HASH_TYPE_SHA256, secret, 32,
                      sinfo, 10, out, 32);
```

C.22.2.100 function wc_ecc_curve_cache_init

```
int wc_ecc_curve_cache_init(
    void
)
```

Initializes curve cache.

See: wc_ecc_curve_cache_free

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_ecc_curve_cache_init();
```

C.22.2.101 function wc_ecc_curve_cache_free

```
void wc_ecc_curve_cache_free(
    void
)
```

Frees curve cache.

See: wc_ecc_curve_cache_init

Return: none No returns

Example

```
wc_ecc_curve_cache_free();
```

C.22.2.102 function wc_ecc_gen_k

```
int wc_ecc_gen_k(
    WC_RNG * rng,
    int size,
    mp_int * k,
    mp_int * order
)
```

Generates random k value.

Parameters:

- **rng** Random number generator
- **size** Key size
- **k** Output k value
- **order** Curve order

See: wc_ecc_sign_hash

Return:

- 0 on success
- negative on error

Example

```
WC_RNG rng;
mp_int k, order;
int ret = wc_ecc_gen_k(&rng, 32, &k, &order);
```

C.22.2.103 function wc_ecc_set_handle

```
int wc_ecc_set_handle(
    ecc_key * key,
    remote_handle64 handle
)
```

Sets remote handle for hardware.

Parameters:

- **key** ECC key
- **handle** Remote handle

See: wc_ecc_init

Return:

- 0 on success
- negative on error

Example

```
ecc_key key;
remote_handle64 handle = 0x1234;
int ret = wc_ecc_set_handle(&key, handle);
```

C.22.2.104 function wc_ecc_use_key_id

```
int wc_ecc_use_key_id(
    ecc_key * key,
    word32 keyId,
    word32 flags
)
```

Uses key ID for hardware.

Parameters:

- **key** ECC key
- **keyId** Key identifier
- **flags** Flags

See: wc_ecc_get_key_id

Return:

- 0 on success
- negative on error

Example

```
ecc_key key;
int ret = wc_ecc_use_key_id(&key, 1, 0);
```

C.22.2.105 function wc_ecc_get_key_id

```
int wc_ecc_get_key_id(
    ecc_key * key,
    word32 * keyId
)
```

Gets key ID from hardware key.

Parameters:

- **key** ECC key
- **keyId** Key identifier pointer

See: wc_ecc_use_key_id

Return:

- 0 on success
- negative on error

Example

```
ecc_key key;
word32 keyId;
int ret = wc_ecc_get_key_id(&key, &keyId);
```

C.22.3 Source code

```
int wc_ecc_make_key(WC_RNG* rng, int keysize, ecc_key* key);

int wc_ecc_make_key_ex(WC_RNG* rng, int keysize, ecc_key* key, int curve_id);

int wc_ecc_make_pub(ecc_key* key, ecc_point* pubOut);
```

```
int wc_ecc_make_pub_ex(ecc_key* key, ecc_point* pubOut, WC_RNG* rng);

int wc_ecc_check_key(ecc_key* key);

void wc_ecc_key_free(ecc_key* key);

int wc_ecc_shared_secret(ecc_key* private_key, ecc_key* public_key, byte* out,
                      word32* outlen);

int wc_ecc_shared_secret_ex(ecc_key* private_key, ecc_point* point,
                            byte* out, word32 *outlen);

int wc_ecc_sign_hash(const byte* in, word32 inlen, byte* out, word32 *outlen,
                     WC_RNG* rng, ecc_key* key);

int wc_ecc_sign_hash_ex(const byte* in, word32 inlen, WC_RNG* rng,
                        ecc_key* key, mp_int *r, mp_int *s);

int wc_ecc_verify_hash(const byte* sig, word32 siglen, const byte* hash,
                    word32 hashlen, int* res, ecc_key* key);

int wc_ecc_verify_hash_ex(mp_int *r, mp_int *s, const byte* hash,
                          word32 hashlen, int* res, ecc_key* key);

int wc_ecc_init(ecc_key* key);

int wc_ecc_init_ex(ecc_key* key, void* heap, int devId);

ecc_key* wc_ecc_key_new(void* heap);

ecc_key* wc_ecc_key_new_ex(void* heap, int devId);

int wc_ecc_free(ecc_key* key);

void wc_ecc_fp_free(void);

int wc_ecc_is_valid_idx(int n);

ecc_point* wc_ecc_new_point(void);

void wc_ecc_del_point(ecc_point* p);

int wc_ecc_copy_point(const ecc_point* p, ecc_point *r);

int wc_ecc_cmp_point(ecc_point* a, ecc_point *b);

int wc_ecc_point_is_at_infinity(ecc_point *p);

int wc_ecc_mulmod(const mp_int* k, ecc_point *G, ecc_point *R,
                  mp_int* a, mp_int* modulus, int map);

int wc_ecc_export_x963(ecc_key* key, byte* out, word32* outLen);
```

```
int wc_ecc_export_x963_ex(ecc_key* key, byte* out, word32* outLen, int
 ↪  compressed);

int wc_ecc_import_x963(const byte* in, word32 inLen, ecc_key* key);

int wc_ecc_import_private_key(const byte* priv, word32 privSz, const byte* pub,
                              word32 pubSz, ecc_key* key);

int wc_ecc_import_wrapped_private(ecc_key* key, int curve_id,
                                  const byte* seed, word32 seedSz,
                                  const byte* wrapped, word32 wrappedLen,
                                  word32 plainLen);

int wc_ecc_import_wrapped_private_ex(ecc_key* key, int curve_id,
                              const byte* wrapped, word32 wrappedLen,
                              const byte* iv, word32 ivLen,
                              const byte* tag, word32 tagLen,
                              const byte* pub, word32 pubLen);

int wc_ecc_rs_to_sig(const char* r, const char* s, byte* out, word32* outlen);

int wc_ecc_import_raw(ecc_key* key, const char* qx, const char* qy,
                      const char* d, const char* curveName);

int wc_ecc_export_private_only(ecc_key* key, byte* out, word32* outLen);

int wc_ecc_export_point_der(const int curve_idx, ecc_point* point,
                            byte* out, word32* outLen);

int wc_ecc_import_point_der(const byte* in, word32 inLen, const int curve_idx,
                            ecc_point* point);

int wc_ecc_size(ecc_key* key);

int wc_ecc_sig_size_calc(int sz);


int wc_ecc_sig_size(const ecc_key* key);


ecEncCtx* wc_ecc_ctx_new(int flags, WC_RNG* rng);

void wc_ecc_ctx_free(ecEncCtx* ctx);

int wc_ecc_ctx_reset(ecEncCtx* ctx, WC_RNG* rng);  /* reset for use again w/o
 ↪  alloc/free */

int wc_ecc_ctx_set_algo(ecEncCtx* ctx, byte encAlgo, byte kdfAlgo,
    byte macAlgo);

int wc_ecc_ctx_set_dev_id(ecEncCtx* ctx, int devId);

int wc_ecc_ctx_get_dev_id(ecEncCtx* ctx, int* devId);
```

```
const byte* wc_ecc_ctx_get_own_salt(ecEncCtx* ctx);

int wc_ecc_ctx_set_peer_salt(ecEncCtx* ctx, const byte* salt);

int wc_ecc_ctx_set_kdf_salt(ecEncCtx* ctx, const byte* salt, word32 sz);

int wc_ecc_ctx_set_info(ecEncCtx* ctx, const byte* info, int sz);

int wc_ecc_encrypt(ecc_key* privKey, ecc_key* pubKey, const byte* msg,
                word32 msgSz, byte* out, word32* outSz, ecEncCtx* ctx);

int wc_ecc_encrypt_ex(ecc_key* privKey, ecc_key* pubKey, const byte* msg,
    word32 msgSz, byte* out, word32* outSz, ecEncCtx* ctx, int compressed);

int wc_ecc_decrypt(ecc_key* privKey, ecc_key* pubKey, const byte* msg,
                word32 msgSz, byte* out, word32* outSz, ecEncCtx* ctx);


int wc_ecc_set_nonblock(ecc_key *key, ecc_nb_ctx_t* ctx);

int wc_ecc_set_curve(ecc_key *key, int keysize, int curve_id);

mp_int* wc_ecc_key_get_priv(ecc_key* key);

size_t wc_ecc_get_sets_count(void);

const char* wc_ecc_get_name(int curve_id);

int wc_ecc_make_key_ex2(WC_RNG* rng, int keysize, ecc_key* key,
    int curve_id, int flags);

int wc_ecc_is_point(ecc_point* ecp, mp_int* a, mp_int* b,
    mp_int* prime);

int wc_ecc_get_generator(ecc_point* ecp, int curve_idx);

int wc_ecc_set_deterministic(ecc_key* key, byte flag);

int wc_ecc_set_deterministic_ex(ecc_key* key, byte flag,
    enum wc_HashType hashType);

int wc_ecc_gen_deterministic_k(const byte* hash, word32 hashSz,
    enum wc_HashType hashType, mp_int* priv, mp_int* k,
    mp_int* order, void* heap);

int wc_ecc_sign_set_k(const byte* k, word32 klen, ecc_key* key);

int wc_ecc_init_id(ecc_key* key, unsigned char* id, int len,
    void* heap, int devId);

int wc_ecc_init_label(ecc_key* key, const char* label, void* heap,
    int devId);

int wc_ecc_set_flags(ecc_key* key, word32 flags);
```

```
void wc_ecc_fp_init(void);

int wc_ecc_set_rng(ecc_key* key, WC_RNG* rng);

int wc_ecc_get_curve_idx(int curve_id);

int wc_ecc_get_curve_id(int curve_idx);

int wc_ecc_get_curve_size_from_id(int curve_id);

int wc_ecc_get_curve_idx_from_name(const char* curveName);

int wc_ecc_get_curve_size_from_name(const char* curveName);

int wc_ecc_get_curve_id_from_name(const char* curveName);

int wc_ecc_get_curve_id_from_params(int fieldSize,
    const byte* prime, word32 primeSz, const byte* Af, word32 AfSz,
    const byte* Bf, word32 BfSz, const byte* order, word32 orderSz,
    const byte* Gx, word32 GxSz, const byte* Gy, word32 GySz,
    int cofactor);

int wc_ecc_get_curve_id_from_dp_params(const ecc_set_type* dp);

int wc_ecc_get_curve_id_from_oid(const byte* oid, word32 len);

const ecc_set_type* wc_ecc_get_curve_params(int curve_idx);

ecc_point* wc_ecc_new_point(void);

ecc_point* wc_ecc_new_point_h(void* h);

void wc_ecc_del_point_h(ecc_point* p, void* h);

void wc_ecc_forcezero_point(ecc_point* p);

int wc_ecc_point_is_on_curve(ecc_point *p, int curve_idx);

int wc_ecc_import_x963_ex(const byte* in, word32 inLen,
    ecc_key* key, int curve_id);

int wc_ecc_import_private_key_ex(const byte* priv, word32 privSz,
    const byte* pub, word32 pubSz, ecc_key* key, int curve_id);

int wc_ecc_rs_raw_to_sig(const byte* r, word32 rSz, const byte* s,
    word32 sSz, byte* out, word32* outlen);

int wc_ecc_sig_to_rs(const byte* sig, word32 sigLen, byte* r,
    word32* rLen, byte* s, word32* sLen);

int wc_ecc_import_raw_ex(ecc_key* key, const char* qx,
    const char* qy, const char* d, int curve_id);
```

```
int wc_ecc_import_unsigned(ecc_key* key, const byte* qx,
    const byte* qy, const byte* d, int curve_id);

int wc_ecc_export_ex(ecc_key* key, byte* qx, word32* qxLen,
    byte* qy, word32* qyLen, byte* d, word32* dLen, int encType);

int wc_ecc_export_public_raw(ecc_key* key, byte* qx,
    word32* qxLen, byte* qy, word32* qyLen);

int wc_ecc_export_private_raw(ecc_key* key, byte* qx,
    word32* qxLen, byte* qy, word32* qyLen, byte* d, word32* dLen);

int wc_ecc_export_point_der_ex(const int curve_idx,
    ecc_point* point, byte* out, word32* outLen, int compressed);

int wc_ecc_import_point_der_ex(const byte* in, word32 inLen,
    const int curve_idx, ecc_point* point, int shortKeySize);

int wc_ecc_get_oid(word32 oidSum, const byte** oid, word32* oidSz);

int wc_ecc_set_custom_curve(ecc_key* key, const ecc_set_type* dp);

ecEncCtx* wc_ecc_ctx_new(int flags, WC_RNG* rng);

ecEncCtx* wc_ecc_ctx_new_ex(int flags, WC_RNG* rng, void* heap);

int wc_ecc_ctx_reset(ecEncCtx* ctx, WC_RNG* rng);

const byte* wc_ecc_ctx_get_own_salt(ecEncCtx* ctx);

int wc_ecc_ctx_set_own_salt(ecEncCtx* ctx, const byte* salt,
    word32 sz);

int wc_X963_KDF(enum wc_HashType type, const byte* secret,
    word32 secretSz, const byte* sinfo, word32 sinfoSz,
    byte* out, word32 outSz);

int wc_ecc_curve_cache_init(void);

void wc_ecc_curve_cache_free(void);

int wc_ecc_gen_k(WC_RNG* rng, int size, mp_int* k, mp_int* order);

int wc_ecc_set_handle(ecc_key* key, remote_handle64 handle);

int wc_ecc_use_key_id(ecc_key* key, word32 keyId, word32 flags);

int wc_ecc_get_key_id(ecc_key* key, word32* keyId);
```

C.23 eccsi.h

C.23.1 Functions

	Name
int	**wc_InitEccsiKey**(EccsiKey * key, void * heap, int devId)
int	**wc_InitEccsiKey_ex**(EccsiKey * key, int keySz, int curveId, void * heap, int devId)
void	**wc_FreeEccsiKey**(EccsiKey * key)
int	**wc_MakeEccsiKey**(EccsiKey * key, WC_RNG * rng)
int	**wc_MakeEccsiPair**(EccsiKey * key, WC_RNG * rng, enum wc_HashType hashType, const byte * id, word32 idSz, mp_int * ssk, ecc_point * pvt)
int	**wc_ValidateEccsiPair**(EccsiKey * key, enum wc_HashType hashType, const byte * id, word32 idSz, const mp_int * ssk, ecc_point * pvt, int * valid)
int	**wc_ValidateEccsiPvt**(EccsiKey * key, const ecc_point * pvt, int * valid)
int	**wc_EncodeEccsiPair**(const EccsiKey * key, mp_int * ssk, ecc_point * pvt, byte * data, word32 * sz)
int	**wc_EncodeEccsiSsk**(const EccsiKey * key, mp_int * ssk, byte * data, word32 * sz)
int	**wc_EncodeEccsiPvt**(const EccsiKey * key, ecc_point * pvt, byte * data, word32 * sz, int raw)
int	**wc_DecodeEccsiPair**(const EccsiKey * key, const byte * data, word32 sz, mp_int * ssk, ecc_point * pvt)
int	**wc_DecodeEccsiSsk**(const EccsiKey * key, const byte * data, word32 sz, mp_int * ssk)
int	**wc_DecodeEccsiPvt**(const EccsiKey * key, const byte * data, word32 sz, ecc_point * pvt)
int	**wc_DecodeEccsiPvtFromSig**(const EccsiKey * key, const byte * sig, word32 sz, ecc_point * pvt)
int	**wc_ExportEccsiKey**(EccsiKey * key, byte * data, word32 * sz)
int	**wc_ImportEccsiKey**(EccsiKey * key, const byte * data, word32 sz)
int	**wc_ExportEccsiPrivateKey**(EccsiKey * key, byte * data, word32 * sz)
int	**wc_ImportEccsiPrivateKey**(EccsiKey * key, const byte * data, word32 sz)
int	**wc_ExportEccsiPublicKey**(EccsiKey * key, byte * data, word32 * sz, int raw)
int	**wc_ImportEccsiPublicKey**(EccsiKey * key, const byte * data, word32 sz, int trusted)
int	**wc_HashEccsiId**(EccsiKey * key, enum wc_HashType hashType, const byte * id, word32 idSz, ecc_point * pvt, byte * hash, byte * hashSz)
int	**wc_SetEccsiHash**(EccsiKey * key, const byte * hash, byte hashSz)

	Name
int	**wc_SetEccsiPair**(EccsiKey * key, const mp_int * ssk, const ecc_point * pvt)
int	**wc_SignEccsiHash**(EccsiKey * key, WC_RNG * rng, enum wc_HashType hashType, const byte * msg, word32 msgSz, byte * sig, word32 * sigSz)
int	**wc_VerifyEccsiHash**(EccsiKey * key, enum wc_HashType hashType, const byte * msg, word32 msgSz, const byte * sig, word32 sigSz, int * verified)

C.23.2 Functions Documentation

C.23.2.1 function wc_InitEccsiKey

```
int wc_InitEccsiKey(
    EccsiKey * key,
    void * heap,
    int devId
)
```

C.23.2.2 function wc_InitEccsiKey_ex

```
int wc_InitEccsiKey_ex(
    EccsiKey * key,
    int keySz,
    int curveId,
    void * heap,
    int devId
)
```

C.23.2.3 function wc_FreeEccsiKey

```
void wc_FreeEccsiKey(
    EccsiKey * key
)
```

C.23.2.4 function wc_MakeEccsiKey

```
int wc_MakeEccsiKey(
    EccsiKey * key,
    WC_RNG * rng
)
```

C.23.2.5 function wc_MakeEccsiPair

```
int wc_MakeEccsiPair(
    EccsiKey * key,
    WC_RNG * rng,
    enum wc_HashType hashType,
    const byte * id,
    word32 idSz,
    mp_int * ssk,
```

```
    ecc_point * pvt
)
```

C.23.2.6 function wc_ValidateEccsiPair

```
int wc_ValidateEccsiPair(
    EccsiKey * key,
    enum wc_HashType hashType,
    const byte * id,
    word32 idSz,
    const mp_int * ssk,
    ecc_point * pvt,
    int * valid
)
```

C.23.2.7 function wc_ValidateEccsiPvt

```
int wc_ValidateEccsiPvt(
    EccsiKey * key,
    const ecc_point * pvt,
    int * valid
)
```

C.23.2.8 function wc_EncodeEccsiPair

```
int wc_EncodeEccsiPair(
    const EccsiKey * key,
    mp_int * ssk,
    ecc_point * pvt,
    byte * data,
    word32 * sz
)
```

C.23.2.9 function wc_EncodeEccsiSsk

```
int wc_EncodeEccsiSsk(
    const EccsiKey * key,
    mp_int * ssk,
    byte * data,
    word32 * sz
)
```

C.23.2.10 function wc_EncodeEccsiPvt

```
int wc_EncodeEccsiPvt(
    const EccsiKey * key,
    ecc_point * pvt,
    byte * data,
    word32 * sz,
    int raw
)
```

C.23.2.11 function wc_DecodeEccsiPair

```
int wc_DecodeEccsiPair(
    const EccsiKey * key,
    const byte * data,
    word32 sz,
    mp_int * ssk,
    ecc_point * pvt
)
```

C.23.2.12 function wc_DecodeEccsiSsk

```
int wc_DecodeEccsiSsk(
    const EccsiKey * key,
    const byte * data,
    word32 sz,
    mp_int * ssk
)
```

C.23.2.13 function wc_DecodeEccsiPvt

```
int wc_DecodeEccsiPvt(
    const EccsiKey * key,
    const byte * data,
    word32 sz,
    ecc_point * pvt
)
```

C.23.2.14 function wc_DecodeEccsiPvtFromSig

```
int wc_DecodeEccsiPvtFromSig(
    const EccsiKey * key,
    const byte * sig,
    word32 sz,
    ecc_point * pvt
)
```

C.23.2.15 function wc_ExportEccsiKey

```
int wc_ExportEccsiKey(
    EccsiKey * key,
    byte * data,
    word32 * sz
)
```

C.23.2.16 function wc_ImportEccsiKey

```
int wc_ImportEccsiKey(
    EccsiKey * key,
    const byte * data,
    word32 sz
)
```

C.23.2.17 function wc_ExportEccsiPrivateKey

```
int wc_ExportEccsiPrivateKey(
    EccsiKey * key,
    byte * data,
    word32 * sz
)
```

C.23.2.18 function wc_ImportEccsiPrivateKey

```
int wc_ImportEccsiPrivateKey(
    EccsiKey * key,
    const byte * data,
    word32 sz
)
```

C.23.2.19 function wc_ExportEccsiPublicKey

```
int wc_ExportEccsiPublicKey(
    EccsiKey * key,
    byte * data,
    word32 * sz,
    int raw
)
```

C.23.2.20 function wc_ImportEccsiPublicKey

```
int wc_ImportEccsiPublicKey(
    EccsiKey * key,
    const byte * data,
    word32 sz,
    int trusted
)
```

C.23.2.21 function wc_HashEccsiId

```
int wc_HashEccsiId(
    EccsiKey * key,
    enum wc_HashType hashType,
    const byte * id,
    word32 idSz,
    ecc_point * pvt,
    byte * hash,
    byte * hashSz
)
```

C.23.2.22 function wc_SetEccsiHash

```
int wc_SetEccsiHash(
    EccsiKey * key,
    const byte * hash,
    byte hashSz
)
```

C.23.2.23 function wc_SetEccsiPair

```
int wc_SetEccsiPair(
    EccsiKey * key,
    const mp_int * ssk,
    const ecc_point * pvt
)
```

C.23.2.24 function wc_SignEccsiHash

```
int wc_SignEccsiHash(
    EccsiKey * key,
    WC_RNG * rng,
    enum wc_HashType hashType,
    const byte * msg,
    word32 msgSz,
    byte * sig,
    word32 * sigSz
)
```

C.23.2.25 function wc_VerifyEccsiHash

```
int wc_VerifyEccsiHash(
    EccsiKey * key,
    enum wc_HashType hashType,
    const byte * msg,
    word32 msgSz,
    const byte * sig,
    word32 sigSz,
    int * verified
)
```

C.23.3 Source code

```
int wc_InitEccsiKey(EccsiKey* key, void* heap, int devId);
int wc_InitEccsiKey_ex(EccsiKey* key, int keySz, int curveId,
        void* heap, int devId);
void wc_FreeEccsiKey(EccsiKey* key);

int wc_MakeEccsiKey(EccsiKey* key, WC_RNG* rng);

int wc_MakeEccsiPair(EccsiKey* key, WC_RNG* rng,
        enum wc_HashType hashType, const byte* id, word32 idSz, mp_int* ssk,
        ecc_point* pvt);
int wc_ValidateEccsiPair(EccsiKey* key, enum wc_HashType hashType,
        const byte* id, word32 idSz, const mp_int* ssk, ecc_point* pvt,
        int* valid);
int wc_ValidateEccsiPvt(EccsiKey* key, const ecc_point* pvt,
        int* valid);
int wc_EncodeEccsiPair(const EccsiKey* key, mp_int* ssk,
        ecc_point* pvt, byte* data, word32* sz);
int wc_EncodeEccsiSsk(const EccsiKey* key, mp_int* ssk, byte* data,
        word32* sz);
int wc_EncodeEccsiPvt(const EccsiKey* key, ecc_point* pvt,
```

```
        byte* data, word32* sz, int raw);
int wc_DecodeEccsiPair(const EccsiKey* key, const byte* data,
        word32 sz, mp_int* ssk, ecc_point* pvt);
int wc_DecodeEccsiSsk(const EccsiKey* key, const byte* data,
        word32 sz, mp_int* ssk);
int wc_DecodeEccsiPvt(const EccsiKey* key, const byte* data,
        word32 sz, ecc_point* pvt);
int wc_DecodeEccsiPvtFromSig(const EccsiKey* key, const byte* sig,
        word32 sz, ecc_point* pvt);

int wc_ExportEccsiKey(EccsiKey* key, byte* data, word32* sz);
int wc_ImportEccsiKey(EccsiKey* key, const byte* data, word32 sz);

int wc_ExportEccsiPrivateKey(EccsiKey* key, byte* data, word32* sz);
int wc_ImportEccsiPrivateKey(EccsiKey* key, const byte* data,
        word32 sz);

int wc_ExportEccsiPublicKey(EccsiKey* key, byte* data, word32* sz,
        int raw);
int wc_ImportEccsiPublicKey(EccsiKey* key, const byte* data,
        word32 sz, int trusted);

int wc_HashEccsiId(EccsiKey* key, enum wc_HashType hashType,
        const byte* id, word32 idSz, ecc_point* pvt, byte* hash, byte* hashSz);
int wc_SetEccsiHash(EccsiKey* key, const byte* hash, byte hashSz);
int wc_SetEccsiPair(EccsiKey* key, const mp_int* ssk,
        const ecc_point* pvt);

int wc_SignEccsiHash(EccsiKey* key, WC_RNG* rng,
        enum wc_HashType hashType, const byte* msg, word32 msgSz, byte* sig,
        word32* sigSz);
int wc_VerifyEccsiHash(EccsiKey* key, enum wc_HashType hashType,
        const byte* msg, word32 msgSz, const byte* sig, word32 sigSz,
        int* verified);
```

C.24 ed25519.h

C.24.1 Functions

	Name
int	**wc_ed25519_make_public**(ed25519_key * key, unsigned char * pubKey, word32 pubKeySz)This function generates the Ed25519 public key from the private key, stored in the ed25519_key object. It stores the public key in the buffer pubKey. If the key object does not already carry a public key, the derived key is also stored in it, so that the key can be used for signing.
int	**wc_ed25519_make_key**(WC_RNG * rng, int keysize, ed25519_key * key)This function generates a new Ed25519 key and stores it in key.

	Name
int	**wc_ed25519_sign_msg**(const byte * in, word32 inlen, byte * out, word32 * outlen, ed25519_key * key)This function signs a message using an ed25519_key object to guarantee authenticity.
int	**wc_ed25519ctx_sign_msg**(const byte * in, word32 inlen, byte * out, word32 * outlen, ed25519_key * key, const byte * context, byte contextLen)This function signs a message using an ed25519_key object to guarantee authenticity. The context is part of the data signed.
int	**wc_ed25519ph_sign_hash**(const byte * hash, word32 hashLen, byte * out, word32 * outLen, ed25519_key * key, const byte * context, byte contextLen)This function signs a message digest using an ed25519_key object to guarantee authenticity. The context is included as part of the data signed. The message is pre-hashed before signature calculation.
int	**wc_ed25519ph_sign_msg**(const byte * in, word32 inlen, byte * out, word32 * outlen, ed25519_key * key, const byte * context, byte contextLen)This function signs a message using an ed25519_key object to guarantee authenticity. The context is included as part of the data signed. The message is pre-hashed before signature calculation.
int	**wc_ed25519_verify_msg**(const byte * sig, word32 siglen, const byte * msg, word32 msgLen, int * ret, ed25519_key * key)This function verifies the Ed25519 signature of a message to ensure authenticity. It returns the answer through ret, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.
int	**wc_ed25519ctx_verify_msg**(const byte * sig, word32 siglen, const byte * msg, word32 msgLen, int * ret, ed25519_key * key, const byte * context, byte contextLen)This function verifies the Ed25519 signature of a message to ensure authenticity. The context is included as part of the data verified. It returns the answer through ret, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

	Name
int	**wc_ed25519ph_verify_hash**(const byte * sig, word32 siglen, const byte * hash, word32 hashLen, int * ret, ed25519_key * key, const byte * context, byte contextLen)This function verifies the Ed25519 signature of the digest of a message to ensure authenticity. The context is included as part of the data verified. The hash is the pre-hashed message before signature calculation. The hash algorithm used to create message digest must be SHA-512. The answer is returned through ret, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.
int	**wc_ed25519ph_verify_msg**(const byte * sig, word32 siglen, const byte * msg, word32 msgLen, int * ret, ed25519_key * key, const byte * context, byte contextLen)This function verifies the Ed25519 signature of a message to ensure authenticity. The context is included as part of the data verified. The message is pre-hashed before verification. It returns the answer through ret, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.
int	**wc_ed25519_init**(ed25519_key * key)This function initializes an ed25519_key object for future use with message verification.
void	**wc_ed25519_free**(ed25519_key * key)This function frees an Ed25519 object after it has been used.
int	**wc_ed25519_import_public**(const byte * in, word32 inLen, ed25519_key * key)This function imports a public ed25519_key from a buffer containing the public key. This function will handle both compressed and uncompressed keys. The public key is checked that it matches the private key when one is present.
int	**wc_ed25519_import_public_ex**(const byte * in, word32 inLen, ed25519_key * key, int trusted)This function imports a public ed25519_key from a buffer containing the public key. This function will handle both compressed and uncompressed keys. Check public key matches private key, when present, when not trusted.
int	**wc_ed25519_import_private_only**(const byte * priv, word32 privSz, ed25519_key * key)This function imports an Ed25519 private key only from a buffer.

	Name
int	**wc_ed25519_import_private_key**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ed25519_key * key)This function imports a public/private Ed25519 key pair from a pair of buffers. This function will handle both compressed and uncompressed keys. The public key is assumed to be untrusted and is checked against the private key.
int	**wc_ed25519_import_private_key_ex**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ed25519_key * key, int trusted)This function imports a public/private Ed25519 key pair from a pair of buffers. This function will handle both compressed and uncompressed keys. The public is checked against private key if not trusted.
int	**wc_ed25519_export_public**(const ed25519_key * key, byte * out, word32 * outLen)This function exports the public key from an ed25519_key structure. It stores the public key in the buffer out, and sets the bytes written to this buffer in outLen.
int	**wc_ed25519_export_private_only**(const ed25519_key * key, byte * out, word32 * outLen)This function exports only the private key from an ed25519_key structure. It stores the private key in the buffer out, and sets the bytes written to this buffer in outLen.
int	**wc_ed25519_export_private**(const ed25519_key * key, byte * out, word32 * outLen)This function exports the key pair from an ed25519_key structure. It stores the key pair in the buffer out, and sets the bytes written to this buffer in outLen.
int	**wc_ed25519_export_key**(const ed25519_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz)This function exports the private and public key separately from an ed25519_key structure. It stores the private key in the buffer priv, and sets the bytes written to this buffer in privSz. It stores the public key in the buffer pub, and sets the bytes written to this buffer in pubSz.
int	**wc_ed25519_check_key**(ed25519_key * key)This function checks the public key in ed25519_key structure matches the private key.
int	**wc_ed25519_size**(const ed25519_key * key)This function returns the size of an Ed25519 - 32 bytes.
int	**wc_ed25519_priv_size**(const ed25519_key * key)This function returns the private key size (secret + public) in bytes.

	Name
int	**wc_ed25519_pub_size**(const ed25519_key * key)This function returns the compressed key size in bytes (public key).
int	**wc_ed25519_sig_size**(const ed25519_key * key)This function returns the size of an Ed25519 signature (64 in bytes).
int	**wc_ed25519_sign_msg_ex**(const byte * in, word32 inLen, byte * out, word32 * outLen, ed25519_key * key, byte type, const byte * context, byte contextLen)Signs message with extended parameters.
int	**wc_ed25519_verify_msg_ex**(const byte * sig, word32 sigLen, const byte * msg, word32 msgLen, int * res, ed25519_key * key, byte type, const byte * context, byte contextLen)Verifies signature with extended parameters.
int	**wc_ed25519_verify_msg_init**(const byte * sig, word32 sigLen, ed25519_key * key, byte type, const byte * context, byte contextLen)Initializes streaming verification.
int	**wc_ed25519_verify_msg_update**(const byte * msgSegment, word32 msgSegmentLen, ed25519_key * key)Updates streaming verification with message segment.
int	**wc_ed25519_verify_msg_final**(const byte * sig, word32 sigLen, int * res, ed25519_key * key)Finalizes streaming verification.
int	**wc_ed25519_init_ex**(ed25519_key * key, void * heap, int devId)Initializes Ed25519 key with extended parameters.
int	**wc_ed25519_init_id**(ed25519_key * key, const unsigned char * id, int len, void * heap, int devId)Initializes an ed25519_key object that references a private key held by a crypto callback device, identified by id. Only available when built with WOLF_PRIVATE_KEY_ID.
int	**wc_ed25519_init_label**(ed25519_key * key, const char * label, void * heap, int devId)Initializes an ed25519_key object that references a private key held by a crypto callback device, identified by label. Only available when built with WOLF_PRIVATE_KEY_ID.
ed25519_key *	**wc_ed25519_new**(void * heap, int devId, int * result_code)Allocates and initializes new Ed25519 key. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.

	Name
int	**wc_ed25519_delete**(ed25519_key * key, ed25519_key ** key_p)Frees and deletes Ed25519 key. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.

C.24.2 Functions Documentation

C.24.2.1 function wc_ed25519_make_public

```
int wc_ed25519_make_public(
    ed25519_key * key,
    unsigned char * pubKey,
    word32 pubKeySz
)
```

This function generates the Ed25519 public key from the private key, stored in the ed25519_key object. It stores the public key in the buffer pubKey. If the key object does not already carry a public key, the derived key is also stored in it, so that the key can be used for signing.

Parameters:

- **key** Pointer to the ed25519_key for which to generate a key.
- **pubKey** Pointer to the buffer in which to store the public key.
- **pubKeySz** Size of the public key. Should be ED25519_PUB_KEY_SIZE.

See:

- wc_ed25519_init
- wc_ed25519_import_private_only
- wc_ed25519_make_key

Return:

- 0 Returned upon successfully making the public key.
- BAD_FUNC_ARG Returned if key or pubKey evaluate to NULL, or if the specified key size is not 32 bytes (Ed25519 has 32 byte keys).
- ECC_PRIV_KEY_E returned if the ed25519_key object does not have the private key in it.
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
int ret;

ed25519_key key;
byte priv[] = { initialize with 32 byte private key };
byte pub[32];
word32 pubSz = sizeof(pub);

wc_ed25519_init(&key);
wc_ed25519_import_private_only(priv, sizeof(priv), &key);
ret = wc_ed25519_make_public(&key, pub, pubSz);
if (ret != 0) {
    // error making public key
}
```

C.24.2.2 function wc_ed25519_make_key

```
int wc_ed25519_make_key(
    WC_RNG * rng,
    int keysize,
    ed25519_key * key
)
```

This function generates a new Ed25519 key and stores it in key.

Parameters:

- **rng** Pointer to an initialized RNG object with which to generate the key.
- **keysize** Length of key to generate. Should always be 32 for Ed25519.
- **key** Pointer to the ed25519_key for which to generate a key.

See: wc_ed25519_init

Return:

- 0 Returned upon successfully making an ed25519_key.
- BAD_FUNC_ARG Returned if rng or key evaluate to NULL, or if the specified key size is not 32 bytes (Ed25519 has 32 byte keys).
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
int ret;

WC_RNG rng;
ed25519_key key;

wc_InitRng(&rng);
wc_ed25519_init(&key);
wc_ed25519_make_key(&rng, 32, &key);
if (ret != 0) {
    // error making key
}
```

C.24.2.3 function wc_ed25519_sign_msg

```
int wc_ed25519_sign_msg(
    const byte * in,
    word32 inlen,
    byte * out,
    word32 * outlen,
    ed25519_key * key
)
```

This function signs a message using an ed25519_key object to guarantee authenticity.

Parameters:

- **in** Pointer to the buffer containing the message to sign.
- **inlen** Length of the message to sign.
- **out** Buffer in which to store the generated signature.
- **outlen** Maximum length of the output buffer. Will store the bytes written to out upon successfully generating a message signature.
- **key** Pointer to a private ed25519_key with which to generate the signature.

See:

- wc_ed25519ctx_sign_msg
- wc_ed25519ph_sign_hash
- wc_ed25519ph_sign_msg
- wc_ed25519_verify_msg

Return:

- 0 Returned upon successfully generating a signature for the message.
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the output buffer is too small to store the generated signature.
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
ed25519_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[64]; // will hold generated signature
sigSz = sizeof(sig);
byte message[] = { initialize with message };

wc_InitRng(&rng); // initialize rng
wc_ed25519_init(&key); // initialize key
wc_ed25519_make_key(&rng, 32, &key); // make public/private key pair
ret = wc_ed25519_sign_msg(message, sizeof(message), sig, &sigSz, &key);
if (ret != 0) {
    // error generating message signature
}
```

C.24.2.4 function wc_ed25519ctx_sign_msg

```
int wc_ed25519ctx_sign_msg(
    const byte * in,
    word32 inlen,
    byte * out,
    word32 * outlen,
    ed25519_key * key,
    const byte * context,
    byte contextLen
)
```

This function signs a message using an ed25519_key object to guarantee authenticity. The context is part of the data signed.

Parameters:

- **in** Pointer to the buffer containing the message to sign.
- **inlen** Length of the message to sign.
- **out** Buffer in which to store the generated signature.
- **outlen** Maximum length of the output buffer. Will store the bytes written to out upon successfully generating a message signature.
- **key** Pointer to a private ed25519_key with which to generate the signature.
- **context** Pointer to the buffer containing the context for which message is being signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed25519_sign_msg

- wc_ed25519ph_sign_hash
- wc_ed25519ph_sign_msg
- wc_ed25519_verify_msg

Return:

- 0 Returned upon successfully generating a signature for the message.
- BAD_FUNC_ARG Returned any of the input parameters evaluate to NULL, or if the output buffer is too small to store the generated signature.
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
ed25519_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[64]; // will hold generated signature
sigSz = sizeof(sig);
byte message[] = { initialize with message };
byte context[] = { initialize with context of signing };

wc_InitRng(&rng); // initialize rng
wc_ed25519_init(&key); // initialize key
wc_ed25519_make_key(&rng, 32, &key); // make public/private key pair
ret = wc_ed25519ctx_sign_msg(message, sizeof(message), sig, &sigSz, &key,
        context, sizeof(context));
if (ret != 0) {
    // error generating message signature
}
```

C.24.2.5 function wc_ed25519ph_sign_hash

```
int wc_ed25519ph_sign_hash(
    const byte * hash,
    word32 hashLen,
    byte * out,
    word32 * outLen,
    ed25519_key * key,
    const byte * context,
    byte contextLen
)
```

This function signs a message digest using an ed25519_key object to guarantee authenticity. The context is included as part of the data signed. The message is pre-hashed before signature calculation.

Parameters:

- **hash** Pointer to the buffer containing the hash of the message to sign.
- **hashLen** Length of the hash of the message to sign.
- **out** Buffer in which to store the generated signature.
- **outLen** Maximum length of the output buffer. Will store the bytes written to out upon successfully generating a message signature.
- **key** Pointer to a private ed25519_key with which to generate the signature.
- **context** Pointer to the buffer containing the context for which message is being signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed25519_sign_msg
- wc_ed25519ctx_sign_msg
- wc_ed25519ph_sign_msg
- wc_ed25519_verify_msg

Return:

- 0 Returned upon successfully generating a signature for the message digest.
- BAD_FUNC_ARG Returned any of the input parameters evaluate to NULL, or if the output buffer is too small to store the generated signature.
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
ed25519_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[64]; // will hold generated signature
sigSz = sizeof(sig);
byte hash[] = { initialize with SHA-512 hash of message };
byte context[] = { initialize with context of signing };

wc_InitRng(&rng); // initialize rng
wc_ed25519_init(&key); // initialize key
wc_ed25519_make_key(&rng, 32, &key); // make public/private key pair
ret = wc_ed25519ph_sign_hash(hash, sizeof(hash), sig, &sigSz, &key,
        context, sizeof(context));
if (ret != 0) {
    // error generating message signature
}
```

C.24.2.6 function wc_ed25519ph_sign_msg

```
int wc_ed25519ph_sign_msg(
    const byte * in,
    word32 inlen,
    byte * out,
    word32 * outlen,
    ed25519_key * key,
    const byte * context,
    byte contextLen
)
```

This function signs a message using an ed25519_key object to guarantee authenticity. The context is included as part of the data signed. The message is pre-hashed before signature calculation.

Parameters:

- **in** Pointer to the buffer containing the message to sign.
- **inlen** Length of the message to sign.
- **out** Buffer in which to store the generated signature.
- **outlen** Maximum length of the output buffer. Will store the bytes written to out upon successfully generating a message signature.
- **key** Pointer to a private ed25519_key with which to generate the signature.
- **context** Pointer to the buffer containing the context for which message is being signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed25519_sign_msg
- wc_ed25519ctx_sign_msg
- wc_ed25519ph_sign_hash
- wc_ed25519_verify_msg

Return:

- 0 Returned upon successfully generating a signature for the message.
- BAD_FUNC_ARG Returned any of the input parameters evaluate to NULL, or if the output buffer is too small to store the generated signature.
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
ed25519_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[64]; // will hold generated signature
sigSz = sizeof(sig);
byte message[] = { initialize with message };
byte context[] = { initialize with context of signing };

wc_InitRng(&rng); // initialize rng
wc_ed25519_init(&key); // initialize key
wc_ed25519_make_key(&rng, 32, &key); // make public/private key pair
ret = wc_ed25519ph_sign_msg(message, sizeof(message), sig, &sigSz, &key,
        context, sizeof(context));
if (ret != 0) {
    // error generating message signature
}
```

C.24.2.7 function wc_ed25519_verify_msg

```
int wc_ed25519_verify_msg(
    const byte * sig,
    word32 siglen,
    const byte * msg,
    word32 msgLen,
    int * ret,
    ed25519_key * key
)
```

This function verifies the Ed25519 signature of a message to ensure authenticity. It returns the answer through ret, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

Parameters:

- **sig** Pointer to the buffer containing the signature to verify.
- **siglen** Length of the signature to verify.
- **msg** Pointer to the buffer containing the message to verify.
- **msgLen** Length of the message to verify.
- **ret** Pointer to the result of the verification. 1 indicates the message was successfully verified.
- **key** Pointer to a public Ed25519 key with which to verify the signature.

See:

- wc_ed25519ctx_verify_msg
- wc_ed25519ph_verify_hash
- wc_ed25519ph_verify_msg
- wc_ed25519_sign_msg

Return:

- 0 Returned upon successfully performing the signature verification and authentication.
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the siglen does not match the actual length of a signature.
- SIG_VERIFY_E Returned if verification completes, but the signature generated does not match the signature provided.

Example

```
ed25519_key key;
int ret, verified = 0;

byte sig[] { initialize with received signature };
byte msg[] = { initialize with message };
// initialize key with received public key
ret = wc_ed25519_verify_msg(sig, sizeof(sig), msg, sizeof(msg), &verified,
        &key);
if (ret < 0) {
    // error performing verification
} else if (verified == 0)
    // the signature is invalid
}
```

C.24.2.8 function wc_ed25519ctx_verify_msg

```
int wc_ed25519ctx_verify_msg(
    const byte * sig,
    word32 siglen,
    const byte * msg,
    word32 msgLen,
    int * ret,
    ed25519_key * key,
    const byte * context,
    byte contextLen
)
```

This function verifies the Ed25519 signature of a message to ensure authenticity. The context is included as part of the data verified. It returns the answer through ret, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

Parameters:

- **sig** Pointer to the buffer containing the signature to verify.
- **siglen** Length of the signature to verify.
- **msg** Pointer to the buffer containing the message to verify.
- **msgLen** Length of the message to verify.
- **ret** Pointer to the result of the verification. 1 indicates the message was successfully verified.
- **key** Pointer to a public Ed25519 key with which to verify the signature.
- **context** Pointer to the buffer containing the context for which the message was signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed25519_verify_msg
- wc_ed25519ph_verify_hash
- wc_ed25519ph_verify_msg
- wc_ed25519_sign_msg

Return:

- 0 Returned upon successfully performing the signature verification and authentication.
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the siglen does not match the actual length of a signature.
- SIG_VERIFY_E Returned if verification completes, but the signature generated does not match the signature provided.

Example

```
ed25519_key key;
int ret, verified = 0;

byte sig[] { initialize with received signature };
byte msg[] = { initialize with message };
byte context[] = { initialize with context of signature };
// initialize key with received public key
ret = wc_ed25519ctx_verify_msg(sig, sizeof(sig), msg, sizeof(msg),
        &verified, &key, );
if (ret < 0) {
    // error performing verification
} else if (verified == 0)
    // the signature is invalid
}
```

C.24.2.9 function wc_ed25519ph_verify_hash

```
int wc_ed25519ph_verify_hash(
    const byte * sig,
    word32 siglen,
    const byte * hash,
    word32 hashLen,
    int * ret,
    ed25519_key * key,
    const byte * context,
    byte contextLen
)
```

This function verifies the Ed25519 signature of the digest of a message to ensure authenticity. The context is included as part of the data verified. The hash is the pre-hashed message before signature calculation. The hash algorithm used to create message digest must be SHA-512. The answer is returned through ret, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

Parameters:

- **sig** Pointer to the buffer containing the signature to verify.
- **siglen** Length of the signature to verify.
- **hash** Pointer to the buffer containing the hash of the message to verify.
- **hashLen** Length of the hash to verify.
- **ret** Pointer to the result of the verification. 1 indicates the message was successfully verified.
- **key** Pointer to a public Ed25519 key with which to verify the signature.
- **context** Pointer to the buffer containing the context for which the message was signed.

- **contextLen** Length of the context buffer.

See:

- wc_ed25519_verify_msg
- wc_ed25519ctx_verify_msg
- wc_ed25519ph_verify_msg
- wc_ed25519_sign_msg

Return:

- 0 Returned upon successfully performing the signature verification and authentication.
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the siglen does not match the actual length of a signature.
- SIG_VERIFY_E Returned if verification completes, but the signature generated does not match the signature provided.

Example

```
ed25519_key key;
int ret, verified = 0;

byte sig[] { initialize with received signature };
byte hash[] = { initialize with SHA-512 hash of message };
byte context[] = { initialize with context of signature };
// initialize key with received public key
ret = wc_ed25519ph_verify_hash(sig, sizeof(sig), hash, sizeof(hash),
        &verified, &key, context, sizeof(context));
if (ret < 0) {
    // error performing verification
} else if (verified == 0)
    // the signature is invalid
}
```

C.24.2.10 function wc_ed25519ph_verify_msg

```
int wc_ed25519ph_verify_msg(
    const byte * sig,
    word32 siglen,
    const byte * msg,
    word32 msgLen,
    int * ret,
    ed25519_key * key,
    const byte * context,
    byte contextLen
)
```

This function verifies the Ed25519 signature of a message to ensure authenticity. The context is included as part of the data verified. The message is pre-hashed before verification. It returns the answer through ret, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

Parameters:

- **sig** Pointer to the buffer containing the signature to verify.
- **siglen** Length of the signature to verify.
- **msg** Pointer to the buffer containing the message to verify.
- **msgLen** Length of the message to verify.
- **ret** Pointer to the result of the verification. 1 indicates the message was successfully verified.

- **key** Pointer to a public Ed25519 key with which to verify the signature.
- **context** Pointer to the buffer containing the context for which the message was signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed25519_verify_msg
- wc_ed25519ctx_verify_msg
- wc_ed25519ph_verify_hash
- wc_ed25519_sign_msg

Return:

- 0 Returned upon successfully performing the signature verification and authentication.
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the siglen does not match the actual length of a signature.
- SIG_VERIFY_E Returned if verification completes, but the signature generated does not match the signature provided.

Example

```
ed25519_key key;
int ret, verified = 0;

byte sig[] { initialize with received signature };
byte msg[] = { initialize with message };
byte context[] = { initialize with context of signature };
// initialize key with received public key
ret = wc_ed25519ph_verify_msg(sig, sizeof(sig), msg, sizeof(msg),
        &verified, &key, context, sizeof(context));
if (ret < 0) {
    // error performing verification
} else if (verified == 0)
    // the signature is invalid
}
```

C.24.2.11 function wc_ed25519_init

```
int wc_ed25519_init(
    ed25519_key * key
)
```

This function initializes an ed25519_key object for future use with message verification.

Parameters:

- **key** Pointer to the ed25519_key object to initialize.

See:

- wc_ed25519_make_key
- wc_ed25519_free

Return:

- 0 Returned upon successfully initializing the ed25519_key object.
- BAD_FUNC_ARG Returned if key is NULL.

Example

```
ed25519_key key;
wc_ed25519_init(&key);
```

C.24.2.12 function wc_ed25519_free

```
void wc_ed25519_free(
    ed25519_key * key
)
```

This function frees an Ed25519 object after it has been used.

Parameters:

- **key** Pointer to the ed25519_key object to free

See: wc_ed25519_init

Example

```
ed25519_key key;
// initialize key and perform secure exchanges
...
wc_ed25519_free(&key);
```

C.24.2.13 function wc_ed25519_import_public

```
int wc_ed25519_import_public(
    const byte * in,
    word32 inLen,
    ed25519_key * key
)
```

This function imports a public ed25519_key from a buffer containing the public key. This function will handle both compressed and uncompressed keys. The public key is checked that it matches the private key when one is present.

Parameters:

- **in** Pointer to the buffer containing the public key.
- **inLen** Length of the buffer containing the public key.
- **key** Pointer to the ed25519_key object in which to store the public key.

See:

- wc_ed25519_import_public_ex
- wc_ed25519_import_private_key
- wc_ed25519_import_private_key_ex
- wc_ed25519_export_public

Return:

- 0 Returned on successfully importing the ed25519_key.
- BAD_FUNC_ARG Returned if in or key evaluate to NULL, or inLen is less than the size of an Ed25519 key.

Example

```
int ret;
byte pub[] = { initialize Ed25519 public key };

ed_25519 key;
wc_ed25519_init_key(&key);
ret = wc_ed25519_import_public(pub, sizeof(pub), &key);
if (ret != 0) {
```

```
    // error importing key
}
```

C.24.2.14 function wc_ed25519_import_public_ex

```
int wc_ed25519_import_public_ex(
    const byte * in,
    word32 inLen,
    ed25519_key * key,
    int trusted
)
```

This function imports a public ed25519_key from a buffer containing the public key. This function will handle both compressed and uncompressed keys. Check public key matches private key, when present, when not trusted.

Parameters:

- **in** Pointer to the buffer containing the public key.
- **inLen** Length of the buffer containing the public key.
- **key** Pointer to the ed25519_key object in which to store the public key.
- **trusted** Public key data is trusted or not.

See:

- wc_ed25519_import_public
- wc_ed25519_import_private_key
- wc_ed25519_import_private_key_ex
- wc_ed25519_export_public

Return:

- 0 Returned on successfully importing the ed25519_key.
- BAD_FUNC_ARG Returned if in or key evaluate to NULL, or inLen is less than the size of an Ed25519 key.

Example

```
int ret;
byte pub[] = { initialize Ed25519 public key };

ed_25519 key;
wc_ed25519_init_key(&key);
ret = wc_ed25519_import_public_ex(pub, sizeof(pub), &key, 1);
if (ret != 0) {
    // error importing key
}
```

C.24.2.15 function wc_ed25519_import_private_only

```
int wc_ed25519_import_private_only(
    const byte * priv,
    word32 privSz,
    ed25519_key * key
)
```

This function imports an Ed25519 private key only from a buffer.

Parameters:

- **priv** Pointer to the buffer containing the private key.
- **privSz** Length of the private key.
- **key** Pointer to the ed25519_key object in which to store the imported private key.

See:

- wc_ed25519_import_public
- wc_ed25519_import_public_ex
- wc_ed25519_import_private_key
- wc_ed25519_import_private_key_ex
- wc_ed25519_export_private_only

Return:

- 0 Returned on successfully importing the Ed25519 key.
- BAD_FUNC_ARG Returned if priv or key evaluate to NULL, or if privSz is not equal to ED25519_KEY_SIZE.

Example

```
int ret;
byte priv[] = { initialize with 32 byte private key };

ed25519_key key;
wc_ed25519_init_key(&key);
ret = wc_ed25519_import_private_only(priv, sizeof(priv), &key);
if (ret != 0) {
    // error importing private key
}
```

C.24.2.16 function wc_ed25519_import_private_key

```
int wc_ed25519_import_private_key(
    const byte * priv,
    word32 privSz,
    const byte * pub,
    word32 pubSz,
    ed25519_key * key
)
```

This function imports a public/private Ed25519 key pair from a pair of buffers. This function will handle both compressed and uncompressed keys. The public key is assumed to be untrusted and is checked against the private key.

Parameters:

- **priv** Pointer to the buffer containing the private key.
- **privSz** Length of the private key.
- **pub** Pointer to the buffer containing the public key.
- **pubSz** Length of the public key.
- **key** Pointer to the ed25519_key object in which to store the imported private/public key pair.

See:

- wc_ed25519_import_public
- wc_ed25519_import_public_ex
- wc_ed25519_import_private_only
- wc_ed25519_import_private_key_ex
- wc_ed25519_export_private

Return:

- 0 Returned on successfully importing the ed25519_key.
- BAD_FUNC_ARG Returned if priv or key evaluate to NULL; or if either privSz is not equal to ED25519_KEY_SIZE nor ED25519_PRV_KEY_SIZE, or pubSz is less than ED25519_PUB_KEY_SIZE.

Example

```
int ret;
byte priv[] = { initialize with 32 byte private key };
byte pub[]  = { initialize with the corresponding public key };

ed25519_key key;
wc_ed25519_init_key(&key);
ret = wc_ed25519_import_private_key(priv, sizeof(priv), pub, sizeof(pub),
        &key);
if (ret != 0) {
    // error importing key
}
```

C.24.2.17 function wc_ed25519_import_private_key_ex

```
int wc_ed25519_import_private_key_ex(
    const byte * priv,
    word32 privSz,
    const byte * pub,
    word32 pubSz,
    ed25519_key * key,
    int trusted
)
```

This function imports a public/private Ed25519 key pair from a pair of buffers. This function will handle both compressed and uncompressed keys. The public is checked against private key if not trusted.

Parameters:

- **priv** Pointer to the buffer containing the private key.
- **privSz** Length of the private key.
- **pub** Pointer to the buffer containing the public key.
- **pubSz** Length of the public key.
- **key** Pointer to the ed25519_key object in which to store the imported private/public key pair.
- **trusted** Public key data is trusted or not.

See:

- wc_ed25519_import_public
- wc_ed25519_import_public_ex
- wc_ed25519_import_private_only
- wc_ed25519_import_private_key
- wc_ed25519_export_private

Return:

- 0 Returned on successfully importing the ed25519_key.
- BAD_FUNC_ARG Returned if priv or key evaluate to NULL; or if either privSz is not equal to ED25519_KEY_SIZE nor ED25519_PRV_KEY_SIZE, or pubSz is less than ED25519_PUB_KEY_SIZE.

Example

```
int ret;
byte priv[] = { initialize with 32 byte private key };
byte pub[]  = { initialize with the corresponding public key };

ed25519_key key;
wc_ed25519_init_key(&key);
ret = wc_ed25519_import_private_key_ex(priv, sizeof(priv), pub, sizeof(pub),
        &key, 1);
if (ret != 0) {
    // error importing key
}
```

C.24.2.18 function wc_ed25519_export_public

```
int wc_ed25519_export_public(
    const ed25519_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports the public key from an ed25519_key structure. It stores the public key in the buffer out, and sets the bytes written to this buffer in outLen.

Parameters:

- **key** Pointer to an ed25519_key structure from which to export the public key.
- **out** Pointer to the buffer in which to store the public key.
- **outLen** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the public key.

See:

- wc_ed25519_import_public
- wc_ed25519_import_public_ex
- wc_ed25519_export_private
- wc_ed25519_export_private_only

Return:

- 0 Returned upon successfully exporting the public key.
- BAD_FUNC_ARG Returned if any of the input values evaluate to NULL.
- BUFFER_E Returned if the buffer provided is not large enough to store the public key. Upon returning this error, the function sets the size required in outLen.

Example

```
int ret;
ed25519_key key;
// initialize key, make key

char pub[32];
word32 pubSz = sizeof(pub);

ret = wc_ed25519_export_public(&key, pub, &pubSz);
if (ret != 0) {
    // error exporting public key
}
```

C.24.2.19 function wc_ed25519_export_private_only

```
int wc_ed25519_export_private_only(
    const ed25519_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports only the private key from an ed25519_key structure. It stores the private key in the buffer out, and sets the bytes written to this buffer in outLen.

Parameters:

- **key** Pointer to an ed25519_key structure from which to export the private key.
- **out** Pointer to the buffer in which to store the private key.
- **outLen** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the private key.

See:

- wc_ed25519_export_public
- wc_ed25519_export_private
- wc_ed25519_import_private_key
- wc_ed25519_import_private_key_ex

Return:

- 0 Returned upon successfully exporting the private key.
- BAD_FUNC_ARG Returned if any of the input values evaluate to NULL.
- BUFFER_E Returned if the buffer provided is not large enough to store the private key.

Example

```
int ret;
ed25519_key key;
// initialize key, make key

char priv[32]; // 32 bytes because only private key
word32 privSz = sizeof(priv);
ret = wc_ed25519_export_private_only(&key, priv, &privSz);
if (ret != 0) {
    // error exporting private key
}
```

C.24.2.20 function wc_ed25519_export_private

```
int wc_ed25519_export_private(
    const ed25519_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports the key pair from an ed25519_key structure. It stores the key pair in the buffer out, and sets the bytes written to this buffer in outLen.

Parameters:

- **key** Pointer to an ed25519_key structure from which to export the key pair.
- **out** Pointer to the buffer in which to store the key pair.
- **outLen** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the key pair.

See:

- wc_ed25519_import_private_key
- wc_ed25519_import_private_key_ex
- wc_ed25519_export_private_only

Return:

- 0 Returned upon successfully exporting the key pair.
- BAD_FUNC_ARG Returned if any of the input values evaluate to NULL.
- BUFFER_E Returned if the buffer provided is not large enough to store the key pair.

Example

```
ed25519_key key;
wc_ed25519_init(&key);

WC_RNG rng;
wc_InitRng(&rng);

wc_ed25519_make_key(&rng, 32, &key); // initialize 32 byte Ed25519 key

byte out[64]; // out needs to be a sufficient buffer size
word32 outLen = sizeof(out);
int key_size = wc_ed25519_export_private(&key, out, &outLen);
if (key_size == BUFFER_E) {
    // Check size of out compared to outLen to see if function reset outLen
}
```

C.24.2.21 function wc_ed25519_export_key

```
int wc_ed25519_export_key(
    const ed25519_key * key,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz
)
```

This function exports the private and public key separately from an ed25519_key structure. It stores the private key in the buffer priv, and sets the bytes written to this buffer in privSz. It stores the public key in the buffer pub, and sets the bytes written to this buffer in pubSz.

Parameters:

- **key** Pointer to an ed25519_key structure from which to export the key pair.
- **priv** Pointer to the buffer in which to store the private key.
- **privSz** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the private key.
- **pub** Pointer to the buffer in which to store the public key.
- **pubSz** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the public key.

See:

- wc_ed25519_export_private
- wc_ed25519_export_public

Return:

- 0 Returned upon successfully exporting the key pair.
- BAD_FUNC_ARG Returned if any of the input values evaluate to NULL.
- BUFFER_E Returned if the buffer provided is not large enough to store the key pair.
- PUBLIC_KEY_E the given key only has a private key present.

Example

```
int ret;
ed25519_key key;
// initialize key, make key

char pub[32];
word32 pubSz = sizeof(pub);
char priv[32];
word32 privSz = sizeof(priv);

ret = wc_ed25519_export_key(&key, priv, &pubSz, pub, &pubSz);
if (ret != 0) {
    // error exporting public key
}
```

C.24.2.22 function wc_ed25519_check_key

```
int wc_ed25519_check_key(
    ed25519_key * key
)
```

This function checks the public key in ed25519_key structure matches the private key.

Parameters:

- **key** Pointer to an ed25519_key structure holding a private and public key.

See:

- wc_ed25519_import_private_key
- wc_ed25519_import_private_key_ex

Return:

- 0 Returned if the private and public key matched.
- BAD_FUNC_ARG Returned if the given key is NULL.
- PUBLIC_KEY_E Returned if the no public key available or is invalid.

Example

```
int ret;
byte priv[] = { initialize with 57 byte private key };
byte pub[]  = { initialize with the corresponding public key };

ed25519_key key;
wc_ed25519_init_key(&key);
wc_ed25519_import_private_key_ex(priv, sizeof(priv), pub, sizeof(pub), &key,
    1);
ret = wc_ed25519_check_key(&key);
if (ret != 0) {
    // error checking key
}
```

C.24.2.23 function wc_ed25519_size

```
int wc_ed25519_size(
    const ed25519_key * key
)
```

This function returns the size of an Ed25519 - 32 bytes.

Parameters:

- **key** Pointer to an ed25519_key structure for which to get the key size.

See: wc_ed25519_make_key

Return:

- ED25519_KEY_SIZE The size of a valid private key (32 bytes).
- BAD_FUNC_ARG Returned if the given key is NULL.

Example

```
int keySz;
ed25519_key key;
// initialize key, make key
keySz = wc_ed25519_size(&key);
if (keySz == 0) {
    // error determining key size
}
```

C.24.2.24 function wc_ed25519_priv_size

```
int wc_ed25519_priv_size(
    const ed25519_key * key
)
```

This function returns the private key size (secret + public) in bytes.

Parameters:

- **key** Pointer to an ed25519_key structure for which to get the key size.

See: wc_ed25519_pub_size

Return:

- ED25519_PRV_KEY_SIZE The size of the private key (64 bytes).
- BAD_FUNC_ARG Returned if key argument is NULL.

Example

```
ed25519_key key;
wc_ed25519_init(&key);

WC_RNG rng;
wc_InitRng(&rng);

wc_ed25519_make_key(&rng, 32, &key); // initialize 32 byte Ed25519 key
int key_size = wc_ed25519_priv_size(&key);
```

C.24.2.25 function wc_ed25519_pub_size

```
int wc_ed25519_pub_size(
    const ed25519_key * key
)
```

This function returns the compressed key size in bytes (public key).

Parameters:

- **key** Pointer to an ed25519_key structure for which to get the key size.

See: wc_ed25519_priv_size

Return:

- ED25519_PUB_KEY_SIZE The size of the compressed public key (32 bytes).
- BAD_FUNC_ARG Returns if key argument is NULL.

Example

```
ed25519_key key;
wc_ed25519_init(&key);
WC_RNG rng;
wc_InitRng(&rng);

wc_ed25519_make_key(&rng, 32, &key); // initialize 32 byte Ed25519 key
int key_size = wc_ed25519_pub_size(&key);
```

C.24.2.26 function wc_ed25519_sig_size

```
int wc_ed25519_sig_size(
    const ed25519_key * key
)
```

This function returns the size of an Ed25519 signature (64 in bytes).

Parameters:

- **key** Pointer to an ed25519_key structure for which to get the signature size.

See: wc_ed25519_sign_msg

Return:

- ED25519_SIG_SIZE The size of an Ed25519 signature (64 bytes).
- BAD_FUNC_ARG Returns if key argument is NULL.

Example

```
int sigSz;
ed25519_key key;
// initialize key, make key

sigSz = wc_ed25519_sig_size(&key);
if (sigSz == 0) {
    // error determining sig size
}
```

C.24.2.27 function wc_ed25519_sign_msg_ex

```
int wc_ed25519_sign_msg_ex(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen,
    ed25519_key * key,
    byte type,
    const byte * context,
    byte contextLen
)
```

Signs message with extended parameters.

Parameters:

- **in** Input message
- **inLen** Input message length
- **out** Output signature buffer
- **outLen** Output signature length pointer
- **key** Ed25519 key
- **type** Signature type
- **context** Context buffer
- **contextLen** Context length

See: wc_ed25519_sign_msg

Return:

- 0 on success
- negative on failure

Example

```
byte msg[] = "message";
byte sig[ED25519_SIG_SIZE];
word32 sigLen = sizeof(sig);
int ret = wc_ed25519_sign_msg_ex(msg, sizeof(msg), sig, &sigLen,
                                 &key, Ed25519, NULL, 0);
```

C.24.2.28 function wc_ed25519_verify_msg_ex

```
int wc_ed25519_verify_msg_ex(
    const byte * sig,
    word32 sigLen,
    const byte * msg,
    word32 msgLen,
    int * res,
    ed25519_key * key,
    byte type,
    const byte * context,
    byte contextLen
)
```

Verifies signature with extended parameters.

Parameters:

- **sig** Signature buffer

- **sigLen** Signature length
- **msg** Message buffer
- **msgLen** Message length
- **res** Verification result pointer
- **key** Ed25519 key
- **type** Signature type
- **context** Context buffer
- **contextLen** Context length

See: wc_ed25519_verify_msg

Return:

- 0 on success
- negative on failure

Example

```
byte msg[] = "message";
byte sig[ED25519_SIG_SIZE];
int res;
int ret = wc_ed25519_verify_msg_ex(sig, sizeof(sig), msg,
                                   sizeof(msg), &res, &key,
                                   Ed25519, NULL, 0);
```

C.24.2.29 function wc_ed25519_verify_msg_init

```
int wc_ed25519_verify_msg_init(
    const byte * sig,
    word32 sigLen,
    ed25519_key * key,
    byte type,
    const byte * context,
    byte contextLen
)
```

Initializes streaming verification.

Parameters:

- **sig** Signature buffer
- **sigLen** Signature length
- **key** Ed25519 key
- **type** Signature type
- **context** Context buffer
- **contextLen** Context length

See:

- wc_ed25519_verify_msg_update
- wc_ed25519_verify_msg_final

Return:

- 0 on success
- negative on failure

Example

```
byte sig[ED25519_SIG_SIZE];
int ret = wc_ed25519_verify_msg_init(sig, sizeof(sig), &key,
                                     Ed25519, NULL, 0);
```

C.24.2.30 function wc_ed25519_verify_msg_update

```
int wc_ed25519_verify_msg_update(
    const byte * msgSegment,
    word32 msgSegmentLen,
    ed25519_key * key
)
```

Updates streaming verification with message segment.

Parameters:

- **msgSegment** Message segment buffer
- **msgSegmentLen** Message segment length
- **key** Ed25519 key

See:

- wc_ed25519_verify_msg_init
- wc_ed25519_verify_msg_final

Return:

- 0 on success
- negative on failure

Example

```
byte msgPart[] = "part";
int ret = wc_ed25519_verify_msg_update(msgPart, sizeof(msgPart),
                                       &key);
```

C.24.2.31 function wc_ed25519_verify_msg_final

```
int wc_ed25519_verify_msg_final(
    const byte * sig,
    word32 sigLen,
    int * res,
    ed25519_key * key
)
```

Finalizes streaming verification.

Parameters:

- **sig** Signature buffer
- **sigLen** Signature length
- **res** Verification result pointer
- **key** Ed25519 key

See:

- wc_ed25519_verify_msg_init
- wc_ed25519_verify_msg_update

Return:

- 0 on success

- negative on failure

Example

```
byte sig[ED25519_SIG_SIZE];
int res;
int ret = wc_ed25519_verify_msg_final(sig, sizeof(sig), &res,
                                      &key);
```

C.24.2.32 function wc_ed25519_init_ex

```
int wc_ed25519_init_ex(
    ed25519_key * key,
    void * heap,
    int devId
)
```

Initializes Ed25519 key with extended parameters.

Parameters:

- **key** Ed25519 key structure
- **heap** Heap hint for memory allocation
- **devId** Device ID for hardware acceleration

See: wc_ed25519_init

Return:

- 0 on success
- negative on failure

Example

```
ed25519_key key;
int ret = wc_ed25519_init_ex(&key, NULL, INVALID_DEVID);
```

C.24.2.33 function wc_ed25519_init_id

```
int wc_ed25519_init_id(
    ed25519_key * key,
    const unsigned char * id,
    int len,
    void * heap,
    int devId
)
```

Initializes an ed25519_key object that references a private key held by a crypto callback device, identified by id. Only available when built with WOLF_PRIVATE_KEY_ID.

Parameters:

- **key** Pointer to the ed25519_key to initialize.
- **id** Key identifier bytes.
- **len** Length of id in bytes.
- **heap** Heap hint. May be NULL.
- **devId** Device identifier for the crypto callback.

See:

- wc_ed25519_init_ex
- wc_ed25519_init_label

Return:

- 0 on success.
- BAD_FUNC_ARG if key is NULL, or id is NULL and len is greater than 0.
- BUFFER_E if len is negative or greater than ED25519_MAX_ID_LEN.

Example

```
ed25519_key key;
unsigned char id[] = { 0x01, 0x02, 0x03, 0x04 };
int ret = wc_ed25519_init_id(&key, id, sizeof(id), NULL, devId);
```

C.24.2.34 function wc_ed25519_init_label

```
int wc_ed25519_init_label(
    ed25519_key * key,
    const char * label,
    void * heap,
    int devId
)
```

Initializes an ed25519_key object that references a private key held by a crypto callback device, identified by label. Only available when built with WOLF_PRIVATE_KEY_ID.

Parameters:

- **key** Pointer to the ed25519_key to initialize.
- **label** NUL-terminated label string.
- **heap** Heap hint. May be NULL.
- **devId** Device identifier for the crypto callback.

See:

- wc_ed25519_init_ex
- wc_ed25519_init_id

Return:

- 0 on success.
- BAD_FUNC_ARG if key or label is NULL.
- BUFFER_E if label is empty or longer than ED25519_MAX_LABEL_LEN.

Example

```
ed25519_key key;
int ret = wc_ed25519_init_label(&key, "my-key", NULL, devId);
```

C.24.2.35 function wc_ed25519_new

```
ed25519_key * wc_ed25519_new(
    void * heap,
    int devId,
    int * result_code
)
```

Allocates and initializes new Ed25519 key. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.

Parameters:

- **heap** Heap hint for memory allocation
- **devId** Device ID for hardware acceleration

- **result_code** Result code pointer

See: wc_ed25519_delete

Return:

- ed25519_key pointer on success
- NULL on failure

Note: This API is only available when WC_NO_CONSTRUCTORS is not defined. WC_NO_CONSTRUCTORS is automatically defined when WOLFSSL_NO_MALLOC is defined.

Example

```
int result;
ed25519_key* key = wc_ed25519_new(NULL, INVALID_DEVID, &result);
```

C.24.2.36 function wc_ed25519_delete

```
int wc_ed25519_delete(
    ed25519_key * key,
    ed25519_key ** key_p
)
```

Frees and deletes Ed25519 key. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.

Parameters:

- **key** Ed25519 key to delete
- **key_p** Pointer to key pointer (set to NULL after delete)

See: wc_ed25519_new

Return:

- 0 on success
- negative on failure

Note: This API is only available when WC_NO_CONSTRUCTORS is not defined. WC_NO_CONSTRUCTORS is automatically defined when WOLFSSL_NO_MALLOC is defined.

Example

```
ed25519_key* key = wc_ed25519_new(NULL, INVALID_DEVID, NULL);
int ret = wc_ed25519_delete(key, &key);
```

C.24.3 Source code

```
int wc_ed25519_make_public(ed25519_key* key, unsigned char* pubKey,
                           word32 pubKeySz);

int wc_ed25519_make_key(WC_RNG* rng, int keysize, ed25519_key* key);

int wc_ed25519_sign_msg(const byte* in, word32 inlen, byte* out,
                        word32 *outlen, ed25519_key* key);

int wc_ed25519ctx_sign_msg(const byte* in, word32 inlen, byte* out,
                        word32 *outlen, ed25519_key* key,
                        const byte* context, byte contextLen);
```

```
int wc_ed25519ph_sign_hash(const byte* hash, word32 hashLen, byte* out,
                           word32 *outLen, ed25519_key* key,
                           const byte* context, byte contextLen);

int wc_ed25519ph_sign_msg(const byte* in, word32 inlen, byte* out,
                          word32 *outlen, ed25519_key* key,
                          const byte* context, byte contextLen);

int wc_ed25519_verify_msg(const byte* sig, word32 siglen, const byte* msg,
                          word32 msgLen, int* ret, ed25519_key* key);

int wc_ed25519ctx_verify_msg(const byte* sig, word32 siglen, const byte* msg,
                             word32 msgLen, int* ret, ed25519_key* key,
                             const byte* context, byte contextLen);

int wc_ed25519ph_verify_hash(const byte* sig, word32 siglen, const byte* hash,
                             word32 hashLen, int* ret, ed25519_key* key,
                             const byte* context, byte contextLen);

int wc_ed25519ph_verify_msg(const byte* sig, word32 siglen, const byte* msg,
                            word32 msgLen, int* ret, ed25519_key* key,
                            const byte* context, byte contextLen);

int wc_ed25519_init(ed25519_key* key);

void wc_ed25519_free(ed25519_key* key);

int wc_ed25519_import_public(const byte* in, word32 inLen, ed25519_key* key);

int wc_ed25519_import_public_ex(const byte* in, word32 inLen, ed25519_key* key,
    int trusted);

int wc_ed25519_import_private_only(const byte* priv, word32 privSz,
                                   ed25519_key* key);

int wc_ed25519_import_private_key(const byte* priv, word32 privSz,
                                  const byte* pub, word32 pubSz, ed25519_key* key);

int wc_ed25519_import_private_key_ex(const byte* priv, word32 privSz,
    const byte* pub, word32 pubSz, ed25519_key* key, int trusted);

int wc_ed25519_export_public(const ed25519_key* key, byte* out, word32*
 ↪  outLen);

int wc_ed25519_export_private_only(const ed25519_key* key, byte* out, word32*
 ↪  outLen);

int wc_ed25519_export_private(const ed25519_key* key, byte* out, word32*
 ↪  outLen);

int wc_ed25519_export_key(const ed25519_key* key,
                          byte* priv, word32 *privSz,
                          byte* pub, word32 *pubSz);
```

```
int wc_ed25519_check_key(ed25519_key* key);

int wc_ed25519_size(const ed25519_key* key);

int wc_ed25519_priv_size(const ed25519_key* key);

int wc_ed25519_pub_size(const ed25519_key* key);

int wc_ed25519_sig_size(const ed25519_key* key);
int wc_ed25519_sign_msg_ex(const byte* in, word32 inLen, byte* out,
    word32 *outLen, ed25519_key* key, byte type, const byte* context,
    byte contextLen);

int wc_ed25519_verify_msg_ex(const byte* sig, word32 sigLen,
    const byte* msg, word32 msgLen, int* res, ed25519_key* key,
    byte type, const byte* context, byte contextLen);

int wc_ed25519_verify_msg_init(const byte* sig, word32 sigLen,
    ed25519_key* key, byte type, const byte* context,
    byte contextLen);

int wc_ed25519_verify_msg_update(const byte* msgSegment,
    word32 msgSegmentLen, ed25519_key* key);

int wc_ed25519_verify_msg_final(const byte* sig, word32 sigLen,
    int* res, ed25519_key* key);

int wc_ed25519_init_ex(ed25519_key* key, void* heap, int devId);

int wc_ed25519_init_id(ed25519_key* key, const unsigned char* id, int len,
    void* heap, int devId);

int wc_ed25519_init_label(ed25519_key* key, const char* label, void* heap,
    int devId);

ed25519_key* wc_ed25519_new(void* heap, int devId, int *result_code);

int wc_ed25519_delete(ed25519_key* key, ed25519_key** key_p);
```

C.25 ed448.h

C.25.1 Functions

	Name
int	**wc_ed448_make_public**(ed448_key * key, unsigned char * pubKey, word32 pubKeySz)This function generates the Ed448 public key from the private key. It stores the public key in the buffer pubKey, and sets the bytes written to this buffer in pubKeySz. If the key object does not already carry a public key, the derived key is also stored in it, so that the key can be used for signing.
int	**wc_ed448_make_key**(WC_RNG * rng, int keysize, ed448_key * key)This function generates a new Ed448 key and stores it in key.
int	**wc_ed448_sign_msg**(const byte * in, word32 inLen, byte * out, word32 * outLen, ed448_key * key, const byte * context, byte contextLen)This function signs a message using an ed448_key object to guarantee authenticity.
int	**wc_ed448ph_sign_hash**(const byte * hash, word32 hashLen, byte * out, word32 * outLen, ed448_key * key, const byte * context, byte contextLen)This function signs a message digest using an ed448_key object to guarantee authenticity. The context is included as part of the data signed. The hash is the pre-hashed message before signature calculation.
int	**wc_ed448ph_sign_msg**(const byte * in, word32 inLen, byte * out, word32 * outLen, ed448_key * key, const byte * context, byte contextLen)This function signs a message using an ed448_key object to guarantee authenticity. The context is included as part of the data signed. The message is pre-hashed before signature calculation.
int	**wc_ed448_verify_msg**(const byte * sig, word32 siglen, const byte * msg, word32 msgLen, int * res, ed448_key * key, const byte * context, byte contextLen)This function verifies the Ed448 signature of a message to ensure authenticity. The context is included as part of the data verified. The answer is returned through res, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

	Name
int	**wc_ed448ph_verify_hash**(const byte * sig, word32 siglen, const byte * hash, word32 hashlen, int * res, ed448_key * key, const byte * context, byte contextLen)This function verifies the Ed448 signature of the digest of a message to ensure authenticity. The context is included as part of the data verified. The hash is the pre-hashed message before signature calculation. The answer is returned through res, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.
int	**wc_ed448ph_verify_msg**(const byte * sig, word32 siglen, const byte * msg, word32 msgLen, int * res, ed448_key * key, const byte * context, byte contextLen)This function verifies the Ed448 signature of a message to ensure authenticity. The context is included as part of the data verified. The message is pre-hashed before verification. The answer is returned through res, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.
int	**wc_ed448_init**(ed448_key * key)This function initializes an ed448_key object for future use with message verification.
int	**wc_ed448_init_id**(ed448_key * key, const unsigned char * id, int len, void * heap, int devId)Initializes an ed448_key object that references a private key held by a crypto callback device, identified by id. Only available when built with WOLF_PRIVATE_KEY_ID.
int	**wc_ed448_init_label**(ed448_key * key, const char * label, void * heap, int devId)Initializes an ed448_key object that references a private key held by a crypto callback device, identified by label. Only available when built with WOLF_PRIVATE_KEY_ID.
void	**wc_ed448_free**(ed448_key * key)This function frees an Ed448 object after it has been used.
int	**wc_ed448_import_public**(const byte * in, word32 inLen, ed448_key * key)This function imports a public ed448_key pair from a buffer containing the public key. This function will handle both compressed and uncompressed keys. The public key is checked that it matches the private key when one is present.

	Name
int	**wc_ed448_import_public_ex**(const byte * in, word32 inLen, ed448_key * key, int trusted)This function imports a public ed448_key pair from a buffer containing the public key. This function will handle both compressed and uncompressed keys. Check public key matches private key, when present, when not trusted.
int	**wc_ed448_import_private_only**(const byte * priv, word32 privSz, ed448_key * key)This function imports an Ed448 private key only from a buffer.
int	**wc_ed448_import_private_key**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ed448_key * key)This function imports a public/private Ed448 key pair from a pair of buffers. This function will handle both compressed and uncompressed keys.
int	**wc_ed448_import_private_key_ex**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ed448_key * key, int trusted)This function imports a public/private Ed448 key pair from a pair of buffers. This function will handle both compressed and uncompressed keys. The public is checked against private key if not trusted.
int	**wc_ed448_export_public**(const ed448_key * key, byte * out, word32 * outLen)This function exports the private key from an ed448_key structure. It stores the public key in the buffer out, and sets the bytes written to this buffer in outLen.
int	**wc_ed448_export_private_only**(const ed448_key * key, byte * out, word32 * outLen)This function exports only the private key from an ed448_key structure. It stores the private key in the buffer out, and sets the bytes written to this buffer in outLen.
int	**wc_ed448_export_private**(const ed448_key * key, byte * out, word32 * outLen)This function exports the key pair from an ed448_key structure. It stores the key pair in the buffer out, and sets the bytes written to this buffer in outLen.
int	**wc_ed448_export_key**(const ed448_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz)This function exports the private and public key separately from an ed448_key structure. It stores the private key in the buffer priv, and sets the bytes written to this buffer in privSz. It stores the public key in the buffer pub, and sets the bytes written to this buffer in pubSz.

	Name
int	**wc_ed448_check_key**(ed448_key * key)This function checks the public key in ed448_key structure matches the private key.
int	**wc_ed448_size**(const ed448_key * key)This function returns the size of an Ed448 private key - 57 bytes.
int	**wc_ed448_priv_size**(const ed448_key * key)This function returns the private key size (secret + public) in bytes.
int	**wc_ed448_pub_size**(const ed448_key * key)This function returns the compressed key size in bytes (public key).
int	**wc_ed448_sig_size**(const ed448_key * key)This function returns the size of an Ed448 signature (114 in bytes).

C.25.2 Functions Documentation

C.25.2.1 function wc_ed448_make_public

```
int wc_ed448_make_public(
    ed448_key * key,
    unsigned char * pubKey,
    word32 pubKeySz
)
```

This function generates the Ed448 public key from the private key. It stores the public key in the buffer pubKey, and sets the bytes written to this buffer in pubKeySz. If the key object does not already carry a public key, the derived key is also stored in it, so that the key can be used for signing.

Parameters:

- **key** Pointer to the ed448_key for which to generate a key.
- **pubKey** Pointer to the buffer in which to store the public key.
- **pubKeySz** Size of the pubKey buffer in bytes.

See:

- wc_ed448_init
- wc_ed448_import_private_only
- wc_ed448_make_key

Return:

- 0 Returned upon successfully making the public key.
- BAD_FUNC_ARG Returned ifi key or pubKey evaluate to NULL, or if the specified key size is not 57 bytes (Ed448 has 57 byte keys).
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
int ret;

ed448_key key;
byte priv[] = { initialize with 57 byte private key };
byte pub[57];
word32 pubSz = sizeof(pub);
```

```
wc_ed448_init(&key);
wc_ed448_import_private_only(priv, sizeof(priv), &key);
ret = wc_ed448_make_public(&key, pub, pubSz);
if (ret != 0) {
    // error making public key
}
```

C.25.2.2 function wc_ed448_make_key

```
int wc_ed448_make_key(
    WC_RNG * rng,
    int keysize,
    ed448_key * key
)
```

This function generates a new Ed448 key and stores it in key.

Parameters:

- **rng** Pointer to an initialized RNG object with which to generate the key.
- **keysize** Length of key to generate. Should always be 57 for Ed448.
- **key** Pointer to the ed448_key for which to generate a key.

See: wc_ed448_init

Return:

- 0 Returned upon successfully making an ed448_key.
- BAD_FUNC_ARG Returned if rng or key evaluate to NULL, or if the specified key size is not 57 bytes (Ed448 has 57 byte keys).
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
int ret;

WC_RNG rng;
ed448_key key;

wc_InitRng(&rng);
wc_ed448_init(&key);
ret = wc_ed448_make_key(&rng, 57, &key);
if (ret != 0) {
    // error making key
}
```

C.25.2.3 function wc_ed448_sign_msg

```
int wc_ed448_sign_msg(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen,
    ed448_key * key,
    const byte * context,
    byte contextLen
)
```

This function signs a message using an ed448_key object to guarantee authenticity.

Parameters:

- **in** Pointer to the buffer containing the message to sign.
- **inLen** Length of the message to sign.
- **out** Buffer in which to store the generated signature.
- **outLen** Maximum length of the output buffer. Will store the bytes written to out upon successfully generating a message signature.
- **key** Pointer to a private ed448_key with which to generate the signature.
- **context** Pointer to the buffer containing the context for which message is being signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed448ph_sign_hash
- wc_ed448ph_sign_msg
- wc_ed448_verify_msg

Return:

- 0 Returned upon successfully generating a signature for the message.
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the output buffer is too small to store the generated signature.
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
ed448_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[114]; // will hold generated signature
sigSz = sizeof(sig);
byte message[] = { initialize with message };

wc_InitRng(&rng); // initialize rng
wc_ed448_init(&key); // initialize key
wc_ed448_make_key(&rng, 57, &key); // make public/private key pair
ret = wc_ed448_sign_msg(message, sizeof(message), sig, &sigSz, &key);
if (ret != 0 ) {
    // error generating message signature
}
```

C.25.2.4 function wc_ed448ph_sign_hash

```
int wc_ed448ph_sign_hash(
    const byte * hash,
    word32 hashLen,
    byte * out,
    word32 * outLen,
    ed448_key * key,
    const byte * context,
    byte contextLen
)
```

This function signs a message digest using an ed448_key object to guarantee authenticity. The context is included as part of the data signed. The hash is the pre-hashed message before signature calculation.

Parameters:

- **hash** Pointer to the buffer containing the hash of the message to sign.
- **hashLen** Length of the hash of the message to sign.
- **out** Buffer in which to store the generated signature.
- **outLen** Maximum length of the output buffer. Will store the bytes written to out upon successfully generating a message signature.
- **key** Pointer to a private ed448_key with which to generate the signature.
- **context** Pointer to the buffer containing the context for which message is being signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed448_sign_msg
- wc_ed448ph_sign_msg
- wc_ed448ph_verify_hash

Return:

- 0 Returned upon successfully generating a signature for the message digest.
- BAD_FUNC_ARG Returned any of the input parameters evaluate to NULL, or if the output buffer is too small to store the generated signature.
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
ed448_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[114]; // will hold generated signature
sigSz = sizeof(sig);
byte hash[] = { initialize hash of message };
byte context[] = { initialize with context of signing };

wc_InitRng(&rng); // initialize rng
wc_ed448_init(&key); // initialize key
wc_ed448_make_key(&rng, 57, &key); // make public/private key pair
ret = wc_ed448ph_sign_hash(hash, sizeof(hash), sig, &sigSz, &key,
        context, sizeof(context));
if (ret != 0) {
    // error generating message signature
}
```

C.25.2.5 function wc_ed448ph_sign_msg

```
int wc_ed448ph_sign_msg(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen,
    ed448_key * key,
    const byte * context,
    byte contextLen
)
```

This function signs a message using an ed448_key object to guarantee authenticity. The context is included as part of the data signed. The message is pre-hashed before signature calculation.

Parameters:

- **in** Pointer to the buffer containing the message to sign.
- **inLen** Length of the message to sign.
- **out** Buffer in which to store the generated signature.
- **outLen** Maximum length of the output buffer. Will store the bytes written to out upon successfully generating a message signature.
- **key** Pointer to a private ed448_key with which to generate the signature.
- **context** Pointer to the buffer containing the context for which message is being signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed448_sign_msg
- wc_ed448ph_sign_hash
- wc_ed448ph_verify_msg

Return:

- 0 Returned upon successfully generating a signature for the message.
- BAD_FUNC_ARG Returned any of the input parameters evaluate to NULL, or if the output buffer is too small to store the generated signature.
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
ed448_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[114]; // will hold generated signature
sigSz = sizeof(sig);
byte message[] = { initialize with message };
byte context[] = { initialize with context of signing };

wc_InitRng(&rng); // initialize rng
wc_ed448_init(&key); // initialize key
wc_ed448_make_key(&rng, 57, &key); // make public/private key pair
ret = wc_ed448ph_sign_msg(message, sizeof(message), sig, &sigSz, &key,
        context, sizeof(context));
if (ret != 0) {
    // error generating message signature
}
```

C.25.2.6 function wc_ed448_verify_msg

```
int wc_ed448_verify_msg(
    const byte * sig,
    word32 siglen,
    const byte * msg,
    word32 msgLen,
    int * res,
    ed448_key * key,
    const byte * context,
    byte contextLen
)
```

This function verifies the Ed448 signature of a message to ensure authenticity. The context is included as part of the data verified. The answer is returned through res, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

Parameters:

- **sig** Pointer to the buffer containing the signature to verify.
- **siglen** Length of the signature to verify.
- **msg** Pointer to the buffer containing the message to verify.
- **msgLen** Length of the message to verify.
- **res** Pointer to an int that will be set to 1 for a valid signature or 0 for an invalid signature after verification completes.
- **key** Pointer to a public Ed448 key with which to verify the signature.
- **context** Pointer to the buffer containing the context for which the message was signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed448ph_verify_hash
- wc_ed448ph_verify_msg
- wc_ed448_sign_msg

Return:

- 0 Returned upon successfully performing the signature verification and authentication.
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the siglen does not match the actual length of a signature.
- SIG_VERIFY_E Returned if verification completes, but the signature generated does not match the signature provided.

Example

```
ed448_key key;
int ret, verified = 0;

byte sig[] { initialize with received signature };
byte msg[] = { initialize with message };
byte context[] = { initialize with context of signature };
// initialize key with received public key
ret = wc_ed448_verify_msg(sig, sizeof(sig), msg, sizeof(msg), &verified,
        &key, context, sizeof(context));
if (ret < 0) {
    // error performing verification
} else if (verified == 0)
    // the signature is invalid
}
```

C.25.2.7 function wc_ed448ph_verify_hash

```
int wc_ed448ph_verify_hash(
    const byte * sig,
    word32 siglen,
    const byte * hash,
    word32 hashlen,
    int * res,
    ed448_key * key,
    const byte * context,
```

```
    byte contextLen
)
```

This function verifies the Ed448 signature of the digest of a message to ensure authenticity. The context is included as part of the data verified. The hash is the pre-hashed message before signature calculation. The answer is returned through res, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

Parameters:

- **sig** Pointer to the buffer containing the signature to verify.
- **siglen** Length of the signature to verify.
- **hash** Pointer to the buffer containing the hash of the message to verify.
- **hashlen** Length of the hash to verify.
- **res** Pointer to an int that will be set to 1 for a valid signature or 0 for an invalid signature after verification completes.
- **key** Pointer to a public Ed448 key with which to verify the signature.
- **context** Pointer to the buffer containing the context for which the message was signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed448_verify_msg
- wc_ed448ph_verify_msg
- wc_ed448ph_sign_hash

Return:

- 0 Returned upon successfully performing the signature verification and authentication.
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the siglen does not match the actual length of a signature.
- SIG_VERIFY_E Returned if verification completes, but the signature generated does not match the signature provided.

Example

```
ed448_key key;
int ret, verified = 0;

byte sig[] { initialize with received signature };
byte hash[] = { initialize hash of message };
byte context[] = { initialize with context of signature };
// initialize key with received public key
ret = wc_ed448ph_verify_hash(sig, sizeof(sig), hash, sizeof(hash),
        &verified, &key, context, sizeof(context));
if (ret < 0) {
    // error performing verification
} else if (verified == 0)
    // the signature is invalid
}
```

C.25.2.8 function wc_ed448ph_verify_msg

```
int wc_ed448ph_verify_msg(
    const byte * sig,
    word32 siglen,
    const byte * msg,
    word32 msgLen,
    int * res,
```

```
    ed448_key * key,
    const byte * context,
    byte contextLen
)
```

This function verifies the Ed448 signature of a message to ensure authenticity. The context is included as part of the data verified. The message is pre-hashed before verification. The answer is returned through res, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

Parameters:

- **sig** Pointer to the buffer containing the signature to verify.
- **siglen** Length of the signature to verify.
- **msg** Pointer to the buffer containing the message to verify.
- **msgLen** Length of the message to verify.
- **res** Pointer to an int that will be set to 1 for a valid signature or 0 for an invalid signature after verification completes.
- **key** Pointer to a public Ed448 key with which to verify the signature.
- **context** Pointer to the buffer containing the context for which the message was signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed448_verify_msg
- wc_ed448ph_verify_hash
- wc_ed448ph_sign_msg

Return:

- 0 Returned upon successfully performing the signature verification and authentication.
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the siglen does not match the actual length of a signature.
- SIG_VERIFY_E Returned if verification completes, but the signature generated does not match the signature provided.

Example

```
ed448_key key;
int ret, verified = 0;

byte sig[] { initialize with received signature };
byte msg[] = { initialize with message };
byte context[] = { initialize with context of signature };
// initialize key with received public key
ret = wc_ed448ph_verify_msg(sig, sizeof(sig), msg, sizeof(msg), &verified,
        &key, context, sizeof(context));
if (ret < 0) {
    // error performing verification
} else if (verified == 0)
    // the signature is invalid
}
```

C.25.2.9 function wc_ed448_init

```
int wc_ed448_init(
    ed448_key * key
)
```

This function initializes an ed448_key object for future use with message verification.

Parameters:

- **key** Pointer to the ed448_key object to initialize.

See:

- wc_ed448_make_key
- wc_ed448_free

Return:

- 0 Returned upon successfully initializing the ed448_key object.
- BAD_FUNC_ARG Returned if key is NULL.

Example

```
ed448_key key;
wc_ed448_init(&key);
```

C.25.2.10 function wc_ed448_init_id

```
int wc_ed448_init_id(
    ed448_key * key,
    const unsigned char * id,
    int len,
    void * heap,
    int devId
)
```

Initializes an ed448_key object that references a private key held by a crypto callback device, identified by id. Only available when built with WOLF_PRIVATE_KEY_ID.

Parameters:

- **key** Pointer to the ed448_key to initialize.
- **id** Key identifier bytes.
- **len** Length of id in bytes.
- **heap** Heap hint. May be NULL.
- **devId** Device identifier for the crypto callback.

See:

- wc_ed448_init_ex
- wc_ed448_init_label

Return:

- 0 on success.
- BAD_FUNC_ARG if key is NULL, or id is NULL and len is greater than 0.
- BUFFER_E if len is negative or greater than ED448_MAX_ID_LEN.

Example

```
ed448_key key;
unsigned char id[] = { 0x01, 0x02, 0x03, 0x04 };
int ret = wc_ed448_init_id(&key, id, sizeof(id), NULL, devId);
```

C.25.2.11 function wc_ed448_init_label

```
int wc_ed448_init_label(
    ed448_key * key,
    const char * label,
```

```
    void * heap,
    int devId
)
```

Initializes an ed448_key object that references a private key held by a crypto callback device, identified by label. Only available when built with WOLF_PRIVATE_KEY_ID.

Parameters:

- **key** Pointer to the ed448_key to initialize.
- **label** NUL-terminated label string.
- **heap** Heap hint. May be NULL.
- **devId** Device identifier for the crypto callback.

See:

- wc_ed448_init_ex
- wc_ed448_init_id

Return:

- 0 on success.
- BAD_FUNC_ARG if key or label is NULL.
- BUFFER_E if label is empty or longer than ED448_MAX_LABEL_LEN.

Example

```
ed448_key key;
int ret = wc_ed448_init_label(&key, "my-key", NULL, devId);
```

C.25.2.12 function wc_ed448_free

```
void wc_ed448_free(
    ed448_key * key
)
```

This function frees an Ed448 object after it has been used.

Parameters:

- **key** Pointer to the ed448_key object to free

See: wc_ed448_init

Example

```
ed448_key key;
// initialize key and perform secure exchanges
...
wc_ed448_free(&key);
```

C.25.2.13 function wc_ed448_import_public

```
int wc_ed448_import_public(
    const byte * in,
    word32 inLen,
    ed448_key * key
)
```

This function imports a public ed448_key pair from a buffer containing the public key. This function will handle both compressed and uncompressed keys. The public key is checked that it matches the private key when one is present.

Parameters:

- **in** Pointer to the buffer containing the public key.
- **inLen** Length of the buffer containing the public key.
- **key** Pointer to the ed448_key object in which to store the public key.

See:

- wc_ed448_import_public_ex
- wc_ed448_import_private_key
- wc_ed448_import_private_key_ex
- wc_ed448_export_public

Return:

- 0 Returned on successfully importing the ed448_key.
- BAD_FUNC_ARG Returned if in or key evaluate to NULL, or inLen is less than the size of an Ed448 key.

Example

```
int ret;
byte pub[] = { initialize Ed448 public key };

ed_448 key;
wc_ed448_init_key(&key);
ret = wc_ed448_import_public(pub, sizeof(pub), &key);
if (ret != 0) {
    // error importing key
}
```

C.25.2.14 function wc_ed448_import_public_ex

```
int wc_ed448_import_public_ex(
    const byte * in,
    word32 inLen,
    ed448_key * key,
    int trusted
)
```

This function imports a public ed448_key pair from a buffer containing the public key. This function will handle both compressed and uncompressed keys. Check public key matches private key, when present, when not trusted.

Parameters:

- **in** Pointer to the buffer containing the public key.
- **inLen** Length of the buffer containing the public key.
- **key** Pointer to the ed448_key object in which to store the public key.
- **trusted** Public key data is trusted or not.

See:

- wc_ed448_import_public
- wc_ed448_import_private_key
- wc_ed448_import_private_key_ex
- wc_ed448_export_public

Return:

- 0 Returned on successfully importing the ed448_key.

- BAD_FUNC_ARG Returned if in or key evaluate to NULL, or inLen is less than the size of an Ed448 key.

Example

```
int ret;
byte pub[] = { initialize Ed448 public key };

ed_448 key;
wc_ed448_init_key(&key);
ret = wc_ed448_import_public_ex(pub, sizeof(pub), &key, 1);
if (ret != 0) {
    // error importing key
}
```

C.25.2.15 function wc_ed448_import_private_only

```
int wc_ed448_import_private_only(
    const byte * priv,
    word32 privSz,
    ed448_key * key
)
```

This function imports an Ed448 private key only from a buffer.

Parameters:

- **priv** Pointer to the buffer containing the private key.
- **privSz** Length of the private key.
- **key** Pointer to the ed448_key object in which to store the imported private key.

See:

- wc_ed448_import_public
- wc_ed448_import_public_ex
- wc_ed448_import_private_key
- wc_ed448_import_private_key_ex
- wc_ed448_export_private_only

Return:

- 0 Returned on successfully importing the Ed448 private key.
- BAD_FUNC_ARG Returned if in or key evaluate to NULL, or if privSz is less than ED448_KEY_SIZE.

Example

```
int ret;
byte priv[] = { initialize with 57 byte private key };

ed448_key key;
wc_ed448_init_key(&key);
ret = wc_ed448_import_private_only(priv, sizeof(priv), &key);
if (ret != 0) {
    // error importing private key
}
```

C.25.2.16 function wc_ed448_import_private_key

```
int wc_ed448_import_private_key(
    const byte * priv,
```

```
    word32 privSz,
    const byte * pub,
    word32 pubSz,
    ed448_key * key
)
```

This function imports a public/private Ed448 key pair from a pair of buffers. This function will handle both compressed and uncompressed keys.

Parameters:

- **priv** Pointer to the buffer containing the private key.
- **privSz** Length of the private key.
- **pub** Pointer to the buffer containing the public key.
- **pubSz** Length of the public key.
- **key** Pointer to the ed448_key object in which to store the imported private/public key pair.

See:

- wc_ed448_import_public
- wc_ed448_import_public_ex
- wc_ed448_import_private_only
- wc_ed448_import_private_key_ex
- wc_ed448_export_private

Return:

- 0 Returned on successfully importing the Ed448 key.
- BAD_FUNC_ARG Returned if in or key evaluate to NULL, or if either privSz is less than ED448_KEY_SIZE or pubSz is less than ED448_PUB_KEY_SIZE.

Example

```
int ret;
byte priv[] = { initialize with 57 byte private key };
byte pub[]  = { initialize with the corresponding public key };

ed448_key key;
wc_ed448_init_key(&key);
ret = wc_ed448_import_private_key(priv, sizeof(priv), pub, sizeof(pub),
        &key);
if (ret != 0) {
    // error importing key
}
```

C.25.2.17 function wc_ed448_import_private_key_ex

```
int wc_ed448_import_private_key_ex(
    const byte * priv,
    word32 privSz,
    const byte * pub,
    word32 pubSz,
    ed448_key * key,
    int trusted
)
```

This function imports a public/private Ed448 key pair from a pair of buffers. This function will handle both compressed and uncompressed keys. The public is checked against private key if not trusted.

Parameters:

- **priv** Pointer to the buffer containing the private key.
- **privSz** Length of the private key.
- **pub** Pointer to the buffer containing the public key.
- **pubSz** Length of the public key.
- **key** Pointer to the ed448_key object in which to store the imported private/public key pair.
- **trusted** Public key data is trusted or not.

See:

- wc_ed448_import_public
- wc_ed448_import_public_ex
- wc_ed448_import_private_only
- wc_ed448_import_private_key
- wc_ed448_export_private

Return:

- 0 Returned on successfully importing the Ed448 key.
- BAD_FUNC_ARG Returned if in or key evaluate to NULL, or if either privSz is less than ED448_KEY_SIZE or pubSz is less than ED448_PUB_KEY_SIZE.

Example

```
int ret;
byte priv[] = { initialize with 57 byte private key };
byte pub[]  = { initialize with the corresponding public key };

ed448_key key;
wc_ed448_init_key(&key);
ret = wc_ed448_import_private_key_ex(priv, sizeof(priv), pub, sizeof(pub),
        &key, 1);
if (ret != 0) {
    // error importing key
}
```

C.25.2.18 function wc_ed448_export_public

```
int wc_ed448_export_public(
    const ed448_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports the private key from an ed448_key structure. It stores the public key in the buffer out, and sets the bytes written to this buffer in outLen.

Parameters:

- **key** Pointer to an ed448_key structure from which to export the public key.
- **out** Pointer to the buffer in which to store the public key.
- **outLen** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the public key.

See:

- wc_ed448_import_public
- wc_ed448_import_public_ex
- wc_ed448_export_private_only

Return:

- 0 Returned upon successfully exporting the public key.
- BAD_FUNC_ARG Returned if any of the input values evaluate to NULL.
- BUFFER_E Returned if the buffer provided is not large enough to store the private key. Upon returning this error, the function sets the size required in outLen.
- PUBLIC_KEY_E the given key only has a private key present.

Example

```
int ret;
ed448_key key;
// initialize key, make key

char pub[57];
word32 pubSz = sizeof(pub);

ret = wc_ed448_export_public(&key, pub, &pubSz);
if (ret != 0) {
    // error exporting public key
}
```

C.25.2.19 function wc_ed448_export_private_only

```
int wc_ed448_export_private_only(
    const ed448_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports only the private key from an ed448_key structure. It stores the private key in the buffer out, and sets the bytes written to this buffer in outLen.

Parameters:

- **key** Pointer to an ed448_key structure from which to export the private key.
- **out** Pointer to the buffer in which to store the private key.
- **outLen** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the private key.

See:

- wc_ed448_export_public
- wc_ed448_import_private_key
- wc_ed448_import_private_key_ex

Return:

- 0 Returned upon successfully exporting the private key.
- ECC_BAD_ARG_E Returned if any of the input values evaluate to NULL.
- BUFFER_E Returned if the buffer provided is not large enough to store the private key.

Example

```
int ret;
ed448_key key;
// initialize key, make key

char priv[57]; // 57 bytes because only private key
word32 privSz = sizeof(priv);
ret = wc_ed448_export_private_only(&key, priv, &privSz);
if (ret != 0) {
```

```
    // error exporting private key
}
```

C.25.2.20 function wc_ed448_export_private

```
int wc_ed448_export_private(
    const ed448_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports the key pair from an ed448_key structure. It stores the key pair in the buffer out, and sets the bytes written to this buffer in outLen.

Parameters:

- **key** Pointer to an ed448_key structure from which to export the key pair.
- **out** Pointer to the buffer in which to store the key pair.
- **outLen** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the key pair.

See:

- wc_ed448_import_private
- wc_ed448_export_private_only

Return:

- 0 Returned upon successfully exporting the key pair.
- ECC_BAD_ARG_E Returned if any of the input values evaluate to NULL.
- BUFFER_E Returned if the buffer provided is not large enough to store the key pair.

Example

```
ed448_key key;
wc_ed448_init(&key);

WC_RNG rng;
wc_InitRng(&rng);

wc_ed448_make_key(&rng, 57, &key); // initialize 57 byte Ed448 key

byte out[114]; // out needs to be a sufficient buffer size
word32 outLen = sizeof(out);
int key_size = wc_ed448_export_private(&key, out, &outLen);
if (key_size == BUFFER_E) {
    // Check size of out compared to outLen to see if function reset outLen
}
```

C.25.2.21 function wc_ed448_export_key

```
int wc_ed448_export_key(
    const ed448_key * key,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz
)
```

This function exports the private and public key separately from an ed448_key structure. It stores the private key in the buffer priv, and sets the bytes written to this buffer in privSz. It stores the public key in the buffer pub, and sets the bytes written to this buffer in pubSz.

Parameters:

- **key** Pointer to an ed448_key structure from which to export the key pair.
- **priv** Pointer to the buffer in which to store the private key.
- **privSz** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the private key.
- **pub** Pointer to the buffer in which to store the public key.
- **pubSz** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the public key.

See:

- wc_ed448_export_private
- wc_ed448_export_public

Return:

- 0 Returned upon successfully exporting the key pair.
- ECC_BAD_ARG_E Returned if any of the input values evaluate to NULL.
- BUFFER_E Returned if the buffer provided is not large enough to store the key pair.
- PUBLIC_KEY_E the given key only has a private key present.

Example

```
int ret;
ed448_key key;
// initialize key, make key

char pub[57];
word32 pubSz = sizeof(pub);
char priv[57];
word32 privSz = sizeof(priv);

ret = wc_ed448_export_key(&key, priv, &pubSz, pub, &pubSz);
if (ret != 0) {
    // error exporting private and public key
}
```

C.25.2.22 function wc_ed448_check_key

```
int wc_ed448_check_key(
    ed448_key * key
)
```

This function checks the public key in ed448_key structure matches the private key.

Parameters:

- **key** Pointer to an ed448_key structure holding a private and public key.

See:

- wc_ed448_import_private_key
- wc_ed448_import_private_key_ex

Return:

- 0 Returned if the private and public key matched.

- BAD_FUNC_ARGS Returned if the given key is NULL.

Example

```
int ret;
byte priv[] = { initialize with 57 byte private key };
byte pub[]  = { initialize with the corresponding public key };

ed448_key key;
wc_ed448_init_key(&key);
wc_ed448_import_private_key_ex(priv, sizeof(priv), pub, sizeof(pub), &key,
    1);
ret = wc_ed448_check_key(&key);
if (ret != 0) {
    // error checking key
}
```

C.25.2.23 function wc_ed448_size

```
int wc_ed448_size(
    const ed448_key * key
)
```

This function returns the size of an Ed448 private key - 57 bytes.

Parameters:

- **key** Pointer to an ed448_key structure for which to get the key size.

See: wc_ed448_make_key

Return:

- ED448_KEY_SIZE The size of a valid private key (57 bytes).
- BAD_FUNC_ARGS Returned if the given key is NULL.

Example

```
int keySz;
ed448_key key;
// initialize key, make key
keySz = wc_ed448_size(&key);
if (keySz == 0) {
    // error determining key size
}
```

C.25.2.24 function wc_ed448_priv_size

```
int wc_ed448_priv_size(
    const ed448_key * key
)
```

This function returns the private key size (secret + public) in bytes.

Parameters:

- **key** Pointer to an ed448_key structure for which to get the key size.

See: wc_ed448_pub_size

Return:

- ED448_PRV_KEY_SIZE The size of the private key (114 bytes).
- BAD_FUNC_ARG Returns if key argument is NULL.

Example

```
ed448_key key;
wc_ed448_init(&key);

WC_RNG rng;
wc_InitRng(&rng);

wc_ed448_make_key(&rng, 57, &key); // initialize 57 byte Ed448 key
int key_size = wc_ed448_priv_size(&key);
```

C.25.2.25 function wc_ed448_pub_size

```
int wc_ed448_pub_size(
    const ed448_key * key
)
```

This function returns the compressed key size in bytes (public key).

Parameters:

- **key** Pointer to an ed448_key structure for which to get the key size.

See: wc_ed448_priv_size

Return:

- ED448_PUB_KEY_SIZE The size of the compressed public key (57 bytes).
- BAD_FUNC_ARG Returns if key argument is NULL.

Example

```
ed448_key key;
wc_ed448_init(&key);
WC_RNG rng;
wc_InitRng(&rng);

wc_ed448_make_key(&rng, 57, &key); // initialize 57 byte Ed448 key
int key_size = wc_ed448_pub_size(&key);
```

C.25.2.26 function wc_ed448_sig_size

```
int wc_ed448_sig_size(
    const ed448_key * key
)
```

This function returns the size of an Ed448 signature (114 in bytes).

Parameters:

- **key** Pointer to an ed448_key structure for which to get the signature size.

See: wc_ed448_sign_msg

Return:

- ED448_SIG_SIZE The size of an Ed448 signature (114 bytes).
- BAD_FUNC_ARG Returns if key argument is NULL.

Example

```
int sigSz;
ed448_key key;
// initialize key, make key

sigSz = wc_ed448_sig_size(&key);
if (sigSz == 0) {
    // error determining sig size
}
```

C.25.3 Source code

```
int wc_ed448_make_public(ed448_key* key, unsigned char* pubKey,
                         word32 pubKeySz);

int wc_ed448_make_key(WC_RNG* rng, int keysize, ed448_key* key);

int wc_ed448_sign_msg(const byte* in, word32 inLen, byte* out,
                      word32 *outLen, ed448_key* key,
                      const byte* context, byte contextLen);

int wc_ed448ph_sign_hash(const byte* hash, word32 hashLen, byte* out,
                         word32 *outLen, ed448_key* key,
                         const byte* context, byte contextLen);

int wc_ed448ph_sign_msg(const byte* in, word32 inLen, byte* out,
                        word32 *outLen, ed448_key* key, const byte* context,
                        byte contextLen);

int wc_ed448_verify_msg(const byte* sig, word32 siglen, const byte* msg,
                        word32 msgLen, int* res, ed448_key* key,
                        const byte* context, byte contextLen);

int wc_ed448ph_verify_hash(const byte* sig, word32 siglen, const byte* hash,
                        word32 hashlen, int* res, ed448_key* key,
                        const byte* context, byte contextLen);

int wc_ed448ph_verify_msg(const byte* sig, word32 siglen, const byte* msg,
                        word32 msgLen, int* res, ed448_key* key,
                        const byte* context, byte contextLen);

int wc_ed448_init(ed448_key* key);

int wc_ed448_init_id(ed448_key* key, const unsigned char* id, int len,
    void* heap, int devId);

int wc_ed448_init_label(ed448_key* key, const char* label, void* heap,
    int devId);

void wc_ed448_free(ed448_key* key);

int wc_ed448_import_public(const byte* in, word32 inLen, ed448_key* key);

int wc_ed448_import_public_ex(const byte* in, word32 inLen, ed448_key* key,
```

```
    int trusted);

int wc_ed448_import_private_only(const byte* priv, word32 privSz,
                                 ed448_key* key);

int wc_ed448_import_private_key(const byte* priv, word32 privSz,
                                const byte* pub, word32 pubSz, ed448_key* key);

int wc_ed448_import_private_key_ex(const byte* priv, word32 privSz,
    const byte* pub, word32 pubSz, ed448_key* key, int trusted);

int wc_ed448_export_public(const ed448_key* key, byte* out, word32* outLen);

int wc_ed448_export_private_only(const ed448_key* key, byte* out,
                                 word32* outLen);

int wc_ed448_export_private(const ed448_key* key, byte* out, word32* outLen);

int wc_ed448_export_key(const ed448_key* key,
                        byte* priv, word32 *privSz,
                        byte* pub, word32 *pubSz);

int wc_ed448_check_key(ed448_key* key);


int wc_ed448_size(const ed448_key* key);

int wc_ed448_priv_size(const ed448_key* key);

int wc_ed448_pub_size(const ed448_key* key);

int wc_ed448_sig_size(const ed448_key* key);
```

C.26 error-crypt.h

C.26.1 Functions

	Name
void	**wc_ErrorString**(int err, char * buff)This function stores the error string for a particular error code in the given buffer.
const char *	**wc_GetErrorString**(int error)This function returns the error string for a particular error code.

C.26.2 Functions Documentation

C.26.2.1 function wc_ErrorString

```
void wc_ErrorString(
    int err,
    char * buff
)
```

This function stores the error string for a particular error code in the given buffer.

Parameters:

- **error** error code for which to get the string
- **buffer** buffer in which to store the error string. Buffer should be at least WOLFSSL_MAX_ERROR_SZ (80 bytes) long

See: wc_GetErrorString

Return: none No returns.

Example

```
char errorMsg[WOLFSSL_MAX_ERROR_SZ];
int err = wc_some_function();

if( err != 0) { // error occurred
    wc_ErrorString(err, errorMsg);
}
```

C.26.2.2 function wc_GetErrorString

```
const char * wc_GetErrorString(
    int error
)
```

This function returns the error string for a particular error code.

Parameters:

- **error** error code for which to get the string

See: wc_ErrorString

Return: string Returns the error string for an error code as a string literal.

Example

```
char * errorMsg;
int err = wc_some_function();

if( err != 0) { // error occurred
    errorMsg = wc_GetErrorString(err);
}
```

C.26.3 Source code

```
void wc_ErrorString(int err, char* buff);

const char* wc_GetErrorString(int error);
```

C.27 evp.h

C.27.1 Functions

	Name
const WOLFSSL_EVP_CIPHER *	**wolfSSL_EVP_des_ede3_ecb**(void)Getter functions for the respective WOLFSSL_EVP_CIPHER pointers. wolfSSL_EVP_init() must be called once in the program first to populate these cipher strings. WOLFSSL_DES_ECB macro must be defined for wolfSSL_EVP_des_ede3_ecb().
const WOLFSSL_EVP_CIPHER *	**wolfSSL_EVP_des_cbc**(void)Getter functions for the respective WOLFSSL_EVP_CIPHER pointers. wolfSSL_EVP_init() must be called once in the program first to populate these cipher strings. WOLFSSL_DES_ECB macro must be defined for wolfSSL_EVP_des_ecb().
int	**wolfSSL_EVP_DigestInit_ex**(WOLFSSL_EVP_MD_CTX * ctx, const WOLFSSL_EVP_MD * type, WOLFSSL_ENGINE * impl)Function for initializing WOLFSSL_EVP_MD_CTX. This function is a wrapper for wolfSSL_EVP_DigestInit() because wolfSSL does not use WOLFSSL_ENGINE.
int	**wolfSSL_EVP_CipherInit_ex**(WOLFSSL_EVP_CIPHER_CTX * ctx, const WOLFSSL_EVP_CIPHER * type, WOLFSSL_ENGINE * impl, const unsigned char * key, const unsigned char * iv, int enc)Function for initializing WOLFSSL_EVP_CIPHER_CTX. This function is a wrapper for wolfSSL_CipherInit() because wolfSSL does not use WOLFSSL_ENGINE.
int	**wolfSSL_EVP_EncryptInit_ex**(WOLFSSL_EVP_CIPHER_CTX * ctx, const WOLFSSL_EVP_CIPHER * type, WOLFSSL_ENGINE * impl, const unsigned char * key, const unsigned char * iv)Function for initializing WOLFSSL_EVP_CIPHER_CTX. This function is a wrapper for wolfSSL_EVP_CipherInit() because wolfSSL does not use WOLFSSL_ENGINE. Sets encrypt flag to be encrypt.
int	**wolfSSL_EVP_DecryptInit_ex**(WOLFSSL_EVP_CIPHER_CTX * ctx, const WOLFSSL_EVP_CIPHER * type, WOLFSSL_ENGINE * impl, const unsigned char * key, const unsigned char * iv)Function for initializing WOLFSSL_EVP_CIPHER_CTX. This function is a wrapper for wolfSSL_EVP_CipherInit() because wolfSSL does not use WOLFSSL_ENGINE. Sets encrypt flag to be decrypt.

	Name
int	**wolfSSL_EVP_CipherUpdate**(WOLFSSL_EVP_CIPHER_CTX * ctx, unsigned char * out, int * outl, const unsigned char * in, int inl)Function for encrypting/decrypting data. In buffer is added to be encrypted or decrypted and out buffer holds the results. outl will be the length of encrypted/decrypted information.
int	**wolfSSL_EVP_CipherFinal**(WOLFSSL_EVP_CIPHER_CTX * ctx, unsigned char * out, int * outl)This function performs the final cipher operations adding in padding. If WOLFSSL_EVP_CIPH_NO_PADDING flag is set in WOLFSSL_EVP_CIPHER_CTX structure then 1 is returned and no encryption/decryption is done. If padding flag is seti padding is added and encrypted when ctx is set to encrypt, padding values are checked when set to decrypt.
int	**wolfSSL_EVP_CIPHER_CTX_set_key_length**(WOLFSSL_EVP_CIPHE * ctx, int keylen)Setter function for WOLFSSL_EVP_CIPHER_CTX structure key length.
int	**wolfSSL_EVP_CIPHER_CTX_block_size**(const WOLFSSL_EVP_CIPHER_CTX * ctx)This is a getter function for the ctx block size.
int	**wolfSSL_EVP_CIPHER_block_size**(const WOLFSSL_EVP_CIPHER * cipher)This is a getter function for the block size of cipher.
void	**wolfSSL_EVP_CIPHER_CTX_set_flags**(WOLFSSL_EVP_CIPHER_CTX * ctx, int flags)Setter function for WOLFSSL_EVP_CIPHER_CTX structure.
void	**wolfSSL_EVP_CIPHER_CTX_clear_flags**(WOLFSSL_EVP_CIPHER_C * ctx, int flags)Clearing function for WOLFSSL_EVP_CIPHER_CTX structure.
int	**wolfSSL_EVP_CIPHER_CTX_set_padding**(WOLFSSL_EVP_CIPHER_ * c, int pad)Setter function for WOLFSSL_EVP_CIPHER_CTX structure to use padding.
unsigned long	**wolfSSL_EVP_CIPHER_CTX_flags**(const WOLFSSL_EVP_CIPHER_CTX * ctx)Getter function for WOLFSSL_EVP_CIPHER_CTX structure. Deprecated v1.1.0.

C.27.2 Functions Documentation

C.27.2.1 function wolfSSL_EVP_des_ede3_ecb

```
const WOLFSSL_EVP_CIPHER * wolfSSL_EVP_des_ede3_ecb(
    void
)
```

Getter functions for the respective WOLFSSL_EVP_CIPHER pointers. wolfSSL_EVP_init() must be called once in the program first to populate these cipher strings. WOLFSSL_DES_ECB macro must be defined for wolfSSL_EVP_des_ede3_ecb().

Parameters:

- **none** No parameters.

See: wolfSSL_EVP_CIPHER_CTX_init

Return: pointer Returns a WOLFSSL_EVP_CIPHER pointer for DES EDE3 operations.

Example

```
printf("block size des ede3 cbc = %d\n",
wolfSSL_EVP_CIPHER_block_size(wolfSSL_EVP_des_ede3_cbc()));
printf("block size des ede3 ecb = %d\n",
wolfSSL_EVP_CIPHER_block_size(wolfSSL_EVP_des_ede3_ecb()));
```

C.27.2.2 function wolfSSL_EVP_des_cbc

```
const WOLFSSL_EVP_CIPHER * wolfSSL_EVP_des_cbc(
    void
)
```

Getter functions for the respective WOLFSSL_EVP_CIPHER pointers. wolfSSL_EVP_init() must be called once in the program first to populate these cipher strings. WOLFSSL_DES_ECB macro must be defined for wolfSSL_EVP_des_ecb().

Parameters:

- **none** No parameters.

See: wolfSSL_EVP_CIPHER_CTX_init

Return: pointer Returns a WOLFSSL_EVP_CIPHER pointer for DES operations.

Example

```
WOLFSSL_EVP_CIPHER* cipher;
cipher = wolfSSL_EVP_des_cbc();
…
```

C.27.2.3 function wolfSSL_EVP_DigestInit_ex

```
int wolfSSL_EVP_DigestInit_ex(
    WOLFSSL_EVP_MD_CTX * ctx,
    const WOLFSSL_EVP_MD * type,
    WOLFSSL_ENGINE * impl
)
```

Function for initializing WOLFSSL_EVP_MD_CTX. This function is a wrapper for wolfSSL_EVP_DigestInit() because wolfSSL does not use WOLFSSL_ENGINE.

Parameters:

- **ctx** structure to initialize.
- **type** type of hash to do, for example SHA.
- **impl** engine to use. N/A for wolfSSL, can be NULL.

See:

- wolfSSL_EVP_MD_CTX_new
- wolfCrypt_Init
- wolfSSL_EVP_MD_CTX_free

Return:

- SSL_SUCCESS If successfully set.
- SSL_FAILURE If not successful.

Example

```
WOLFSSL_EVP_MD_CTX* md = NULL;
wolfCrypt_Init();
md = wolfSSL_EVP_MD_CTX_new();
if (md == NULL) {
    printf("error setting md\n");
    return -1;
}
printf("cipher md init ret = %d\n", wolfSSL_EVP_DigestInit_ex(md,
wolfSSL_EVP_sha1(), e));
//free resources
```

C.27.2.4 function wolfSSL_EVP_CipherInit_ex

```
int wolfSSL_EVP_CipherInit_ex(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    const WOLFSSL_EVP_CIPHER * type,
    WOLFSSL_ENGINE * impl,
    const unsigned char * key,
    const unsigned char * iv,
    int enc
)
```

Function for initializing WOLFSSL_EVP_CIPHER_CTX. This function is a wrapper for wolfSSL_CipherInit() because wolfSSL does not use WOLFSSL_ENGINE.

Parameters:

- **ctx** structure to initialize.
- **type** type of encryption/decryption to do, for example AES.
- **impl** engine to use. N/A for wolfSSL, can be NULL.
- **key** key to set .
- **iv** iv if needed by algorithm.
- **enc** encryption (1) or decryption (0) flag.

See:

- wolfSSL_EVP_CIPHER_CTX_new
- wolfCrypt_Init
- wolfSSL_EVP_CIPHER_CTX_free

Return:

- SSL_SUCCESS If successfully set.
- SSL_FAILURE If not successful.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx = NULL;
WOLFSSL_ENGINE* e = NULL;
unsigned char key[16];
unsigned char iv[12];
wolfCrypt_Init();
ctx = wolfSSL_EVP_CIPHER_CTX_new();
if (ctx == NULL) {
    printf("issue creating ctx\n");
```

```
    return -1;
}

printf("cipher init ex error ret = %d\n", wolfSSL_EVP_CipherInit_ex(NULL,
EVP_aes_128_    cbc(), e, key, iv, 1));
printf("cipher init ex success ret = %d\n", wolfSSL_EVP_CipherInit_ex(ctx,
EVP_aes_128_c    bc(), e, key, iv, 1));
// free resources
```

C.27.2.5 function wolfSSL_EVP_EncryptInit_ex

```
int wolfSSL_EVP_EncryptInit_ex(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    const WOLFSSL_EVP_CIPHER * type,
    WOLFSSL_ENGINE * impl,
    const unsigned char * key,
    const unsigned char * iv
)
```

Function for initializing WOLFSSL_EVP_CIPHER_CTX. This function is a wrapper for wolfSSL_EVP_CipherInit() because wolfSSL does not use WOLFSSL_ENGINE. Sets encrypt flag to be encrypt.

Parameters:

- **ctx** structure to initialize.
- **type** type of encryption to do, for example AES.
- **impl** engine to use. N/A for wolfSSL, can be NULL.
- **key** key to use.
- **iv** iv to use.

See:

- wolfSSL_EVP_CIPHER_CTX_new
- wolfCrypt_Init
- wolfSSL_EVP_CIPHER_CTX_free

Return:

- SSL_SUCCESS If successfully set.
- SSL_FAILURE If not successful.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx = NULL;
wolfCrypt_Init();
ctx = wolfSSL_EVP_CIPHER_CTX_new();
if (ctx == NULL) {
    printf("error setting ctx\n");
    return -1;
}
printf("cipher ctx init ret = %d\n", wolfSSL_EVP_EncryptInit_ex(ctx,
wolfSSL_EVP_aes_128_cbc(), e, key, iv));
//free resources
```

C.27.2.6 function wolfSSL_EVP_DecryptInit_ex

```
int wolfSSL_EVP_DecryptInit_ex(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    const WOLFSSL_EVP_CIPHER * type,
```

```
    WOLFSSL_ENGINE * impl,
    const unsigned char * key,
    const unsigned char * iv
)
```

Function for initializing WOLFSSL_EVP_CIPHER_CTX. This function is a wrapper for wolfSSL_EVP_CipherInit() because wolfSSL does not use WOLFSSL_ENGINE. Sets encrypt flag to be decrypt.

Parameters:

- **ctx** structure to initialize.
- **type** type of encryption/decryption to do, for example AES.
- **impl** engine to use. N/A for wolfSSL, can be NULL.
- **key** key to set .
- **iv** iv if needed by algorithm.
- **enc** encryption (1) or decryption (0) flag.

See:

- wolfSSL_EVP_CIPHER_CTX_new
- wolfCrypt_Init
- wolfSSL_EVP_CIPHER_CTX_free

Return:

- SSL_SUCCESS If successfully set.
- SSL_FAILURE If not successful.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx = NULL;
WOLFSSL_ENGINE* e = NULL;
unsigned char key[16];
unsigned char iv[12];

wolfCrypt_Init();

ctx = wolfSSL_EVP_CIPHER_CTX_new();
if (ctx == NULL) {
    printf("issue creating ctx\n");
    return -1;
}

printf("cipher init ex error ret = %d\n", wolfSSL_EVP_DecryptInit_ex(NULL,
EVP_aes_128_    cbc(), e, key, iv, 1));
printf("cipher init ex success ret = %d\n", wolfSSL_EVP_DecryptInit_ex(ctx,
EVP_aes_128_c    bc(), e, key, iv, 1));
// free resources
```

C.27.2.7 function wolfSSL_EVP_CipherUpdate

```
int wolfSSL_EVP_CipherUpdate(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    unsigned char * out,
    int * outl,
    const unsigned char * in,
    int inl
)
```

Function for encrypting/decrypting data. In buffer is added to be encrypted or decrypted and out buffer holds the results. outl will be the length of encrypted/decrypted information.

Parameters:

- **ctx** structure to get cipher type from.
- **out** buffer to hold output, at least inl plus the cipher block size bytes. It must not overlap in.
- **outl** adjusted to be size of output.
- **in** buffer to perform operation on.
- **inl** length of input buffer.

See:

- wolfSSL_EVP_CIPHER_CTX_new
- wolfCrypt_Init
- wolfSSL_EVP_CIPHER_CTX_free

Return:

- SSL_SUCCESS If successful.
- SSL_FAILURE If not successful.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx = NULL;
unsigned char out[100 + EVP_MAX_BLOCK_LENGTH];
int outl;
unsigned char in[100];
int inl = 100;

ctx = wolfSSL_EVP_CIPHER_CTX_new();
// set up ctx
ret = wolfSSL_EVP_CipherUpdate(ctx, out, outl, in, inl);
// check ret value
// buffer out holds outl bytes of data
// free resources
```

C.27.2.8 function wolfSSL_EVP_CipherFinal

```
int wolfSSL_EVP_CipherFinal(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    unsigned char * out,
    int * outl
)
```

This function performs the final cipher operations adding in padding. If WOLFSSL_EVP_CIPH_NO_PADDING flag is set in WOLFSSL_EVP_CIPHER_CTX structure then 1 is returned and no encryption/decryption is done. If padding flag is seti padding is added and encrypted when ctx is set to encrypt, padding values are checked when set to decrypt.

Parameters:

- **ctx** structure to decrypt/encrypt with.
- **out** buffer for final decrypt/encrypt.
- **out1** size of out buffer when data has been added by function.

See: wolfSSL_EVP_CIPHER_CTX_new

Return:

- 1 Returned on success.

- 0 If encountering a failure.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
int out1;
unsigned char out[64];
// create ctx
wolfSSL_EVP_CipherFinal(ctx, out, &out1);
```

C.27.2.9 function wolfSSL_EVP_CIPHER_CTX_set_key_length

```
int wolfSSL_EVP_CIPHER_CTX_set_key_length(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    int keylen
)
```

Setter function for WOLFSSL_EVP_CIPHER_CTX structure key length.

Parameters:

- **ctx** structure to set key length.
- **keylen** key length.

See: wolfSSL_EVP_CIPHER_flags

Return:

- SSL_SUCCESS If successfully set.
- SSL_FAILURE If failed to set key length.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
int keylen;
// create ctx
wolfSSL_EVP_CIPHER_CTX_set_key_length(ctx, keylen);
```

C.27.2.10 function wolfSSL_EVP_CIPHER_CTX_block_size

```
int wolfSSL_EVP_CIPHER_CTX_block_size(
    const WOLFSSL_EVP_CIPHER_CTX * ctx
)
```

This is a getter function for the ctx block size.

Parameters:

- **ctx** the cipher ctx to get block size of.

See: wolfSSL_EVP_CIPHER_block_size

Return: size Returns ctx->block_size.

Example

```
const WOLFSSL_CVP_CIPHER_CTX* ctx;
//set up ctx
printf("block size = %d\n", wolfSSL_EVP_CIPHER_CTX_block_size(ctx));
```

C.27.2.11 function wolfSSL_EVP_CIPHER_block_size

```
int wolfSSL_EVP_CIPHER_block_size(
    const WOLFSSL_EVP_CIPHER * cipher
)
```

This is a getter function for the block size of cipher.

Parameters:

- **cipher** cipher to get block size of.

See: wolfSSL_EVP_aes_256_ctr

Return: size returns the block size.

Example

```
printf("block size = %d\n",
wolfSSL_EVP_CIPHER_block_size(wolfSSL_EVP_aes_256_ecb()));
```

C.27.2.12 function wolfSSL_EVP_CIPHER_CTX_set_flags

```
void wolfSSL_EVP_CIPHER_CTX_set_flags(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    int flags
)
```

Setter function for WOLFSSL_EVP_CIPHER_CTX structure.

Parameters:

- **ctx** structure to set flag.
- **flag** flag to set in structure.

See:

- wolfSSL_EVP_CIPHER_flags
- wolfSSL_EVP_CIPHER_CTX_flags

Return: none No returns.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
int flag;
// create ctx
wolfSSL_EVP_CIPHER_CTX_set_flags(ctx, flag);
```

C.27.2.13 function wolfSSL_EVP_CIPHER_CTX_clear_flags

```
void wolfSSL_EVP_CIPHER_CTX_clear_flags(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    int flags
)
```

Clearing function for WOLFSSL_EVP_CIPHER_CTX structure.

Parameters:

- **ctx** structure to clear flag.
- **flag** flag value to clear in structure.

See:

- wolfSSL_EVP_CIPHER_flags
- wolfSSL_EVP_CIPHER_CTX_flags

Return: none No returns.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
int flag;
// create ctx
wolfSSL_EVP_CIPHER_CTX_clear_flags(ctx, flag);
```

C.27.2.14 function wolfSSL_EVP_CIPHER_CTX_set_padding

```
int wolfSSL_EVP_CIPHER_CTX_set_padding(
    WOLFSSL_EVP_CIPHER_CTX * c,
    int pad
)
```

Setter function for WOLFSSL_EVP_CIPHER_CTX structure to use padding.

Parameters:

- **ctx** structure to set padding flag.
- **padding** 0 for not setting padding, 1 for setting padding.

See: wolfSSL_EVP_CIPHER_CTX_new

Return:

- SSL_SUCCESS If successfully set.
- BAD_FUNC_ARG If null argument passed in.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
// create ctx
wolfSSL_EVP_CIPHER_CTX_set_padding(ctx, 1);
```

C.27.2.15 function wolfSSL_EVP_CIPHER_CTX_flags

```
unsigned long wolfSSL_EVP_CIPHER_CTX_flags(
    const WOLFSSL_EVP_CIPHER_CTX * ctx
)
```

Getter function for WOLFSSL_EVP_CIPHER_CTX structure. Deprecated v1.1.0.

Parameters:

- **ctx** structure to get flag.

See:

- wolfSSL_EVP_CIPHER_CTX_new
- wolfSSL_EVP_CIPHER_flags

Return: unsigned long of flags/mode.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
unsigned long flags;
ctx = wolfSSL_EVP_CIPHER_CTX_new()
flags = wolfSSL_EVP_CIPHER_CTX_flags(ctx);
```

C.27.3 Source code

```
const WOLFSSL_EVP_CIPHER* wolfSSL_EVP_des_ede3_ecb(void);

const WOLFSSL_EVP_CIPHER* wolfSSL_EVP_des_cbc(void);

int wolfSSL_EVP_DigestInit_ex(WOLFSSL_EVP_MD_CTX* ctx,
                                    const WOLFSSL_EVP_MD* type,
                                    WOLFSSL_ENGINE *impl);

int  wolfSSL_EVP_CipherInit_ex(WOLFSSL_EVP_CIPHER_CTX* ctx,
                                   const WOLFSSL_EVP_CIPHER* type,
                                   WOLFSSL_ENGINE *impl,
                                   const unsigned char* key,
                                   const unsigned char* iv,
                                   int enc);

int  wolfSSL_EVP_EncryptInit_ex(WOLFSSL_EVP_CIPHER_CTX* ctx,
                                   const WOLFSSL_EVP_CIPHER* type,
                                   WOLFSSL_ENGINE *impl,
                                   const unsigned char* key,
                                   const unsigned char* iv);

int  wolfSSL_EVP_DecryptInit_ex(WOLFSSL_EVP_CIPHER_CTX* ctx,
                                   const WOLFSSL_EVP_CIPHER* type,
                                   WOLFSSL_ENGINE *impl,
                                   const unsigned char* key,
                                   const unsigned char* iv);

int wolfSSL_EVP_CipherUpdate(WOLFSSL_EVP_CIPHER_CTX *ctx,
                                  unsigned char *out, int *outl,
                                  const unsigned char *in, int inl);

int  wolfSSL_EVP_CipherFinal(WOLFSSL_EVP_CIPHER_CTX *ctx,
                                  unsigned char *out, int *outl);

int  wolfSSL_EVP_CIPHER_CTX_set_key_length(WOLFSSL_EVP_CIPHER_CTX* ctx,
                                                    int keylen);

int wolfSSL_EVP_CIPHER_CTX_block_size(const WOLFSSL_EVP_CIPHER_CTX *ctx);

int wolfSSL_EVP_CIPHER_block_size(const WOLFSSL_EVP_CIPHER *cipher);

void wolfSSL_EVP_CIPHER_CTX_set_flags(WOLFSSL_EVP_CIPHER_CTX *ctx, int flags);

void wolfSSL_EVP_CIPHER_CTX_clear_flags(WOLFSSL_EVP_CIPHER_CTX *ctx, int
 ↪  flags);

int  wolfSSL_EVP_CIPHER_CTX_set_padding(WOLFSSL_EVP_CIPHER_CTX *c, int pad);


unsigned long wolfSSL_EVP_CIPHER_CTX_flags(const WOLFSSL_EVP_CIPHER_CTX *ctx);
```

C.28 hash.h

C.28.1 Functions

	Name
int	**wc_HashGetOID**(enum wc_HashType hash_type)This function will return the OID for the wc_HashType provided.
int	**wc_HashGetDigestSize**(enum wc_HashType hash_type)This function returns the size of the digest (output) for a hash_type. The returns size is used to make sure the output buffer provided to wc_Hash is large enough.
int	**wc_Hash**(enum wc_HashType hash_type, const byte * data, word32 data_len, byte * hash, word32 hash_len)This function performs a hash on the provided data buffer and returns it in the hash buffer provided.
int	**wc_HashUpdate**(wc_HashAlg * hash, enum wc_HashType type, const byte * data, word32 dataSz)This function updates the hash in a wc_HashAlg object with the provided data. The hash algorithm used is the one stored in the wc_HashAlg object when it was initialized with wc_HashInit() or wc_HashInit_ex().
int	**wc_HashFinal**(wc_HashAlg * hash, enum wc_HashType type, byte * out)This function finalizes the hash in a wc_HashAlg object and writes the resulting digest to out. The hash algorithm used is the one stored in the wc_HashAlg object when it was initialized with wc_HashInit() or wc_HashInit_ex().
int	**wc_HashFree**(wc_HashAlg * hash, enum wc_HashType type)This function frees any resources held by a wc_HashAlg object. The hash algorithm used is the one stored in the wc_HashAlg object when it was initialized with wc_HashInit() or wc_HashInit_ex(). The wc_HashAlg object itself is not freed.
int	**wc_HashSetFlags**(wc_HashAlg * hash, enum wc_HashType type, word32 flags)This function sets the flags of the underlying hash object in a wc_HashAlg object. The hash algorithm used is the one stored in the wc_HashAlg object when it was initialized with wc_HashInit() or wc_HashInit_ex(). Available when WOLFSSL_HASH_FLAGS is defined.

	Name
int	**wc_HashGetFlags**(wc_HashAlg * hash, enum wc_HashType type, word32 * flags)This function gets the flags of the underlying hash object in a wc_HashAlg object. The hash algorithm used is the one stored in the wc_HashAlg object when it was initialized with wc_HashInit() or wc_HashInit_ex(). Available when WOLFSSL_HASH_FLAGS is defined.
int	**wc_Md5Hash**(const byte * data, word32 len, byte * hash)Convenience function, handles all the hashing and places the result into hash.
int	**wc_ShaHash**(const byte * data, word32 len, byte * hash)Convenience function, handles all the hashing and places the result into hash.
int	**wc_Sha224Hash**(const byte * data, word32 len, byte * hash)Convenience function, handles all the hashing and places the result into hash.
int	**wc_Sha256Hash**(const byte * data, word32 len, byte * hash)Convenience function, handles all the hashing and places the result into hash.
int	**wc_Sha384Hash**(const byte * data, word32 len, byte * hash)Convenience function, handles all the hashing and places the result into hash.
int	**wc_Sha512Hash**(const byte * data, word32 len, byte * hash)Convenience function, handles all the hashing and places the result into hash.
int	**wc_Sha3_224Hash**(const byte * data, word32 len, byte * hash)Convenience function, handles all the hashing and places the result into hash.
int	**wc_Sha3_256Hash**(const byte * data, word32 len, byte * hash)Convenience function, handles all the hashing and places the result into hash.
int	**wc_Sha3_384Hash**(const byte * data, word32 len, byte * hash)Convenience function, handles all the hashing and places the result into hash.
int	**wc_Sha3_512Hash**(const byte * data, word32 len, byte * hash)Convenience function, handles all the hashing and places the result into hash.
int	**wc_Shake128Hash**(const byte * data, word32 len, byte * hash, word32 hashLen)Convenience function, handles all the hashing and places the result into hash.
int	**wc_Shake256Hash**(const byte * data, word32 len, byte * hash, word32 hashLen)Convenience function, handles all the hashing and places the result into hash.

C.28.2 Functions Documentation

C.28.2.1 function wc_HashGetOID

```
int wc_HashGetOID(
    enum wc_HashType hash_type
)
```

This function will return the OID for the wc_HashType provided.

Parameters:

- **hash_type** A hash type from the “enum wc_HashType” such as “WC_HASH_TYPE_SHA256”.

See:

- wc_HashGetDigestSize
- wc_Hash

Return:

- OID returns value greater than 0
- HASH_TYPE_E hash type not supported.
- BAD_FUNC_ARG one of the provided arguments is incorrect.

Example

```
enum wc_HashType hash_type = WC_HASH_TYPE_SHA256;
int oid = wc_HashGetOID(hash_type);
if (oid > 0) {
    // Success
}
```

C.28.2.2 function wc_HashGetDigestSize

```
int wc_HashGetDigestSize(
    enum wc_HashType hash_type
)
```

This function returns the size of the digest (output) for a hash_type. The returns size is used to make sure the output buffer provided to wc_Hash is large enough.

Parameters:

- **hash_type** A hash type from the “enum wc_HashType” such as “WC_HASH_TYPE_SHA256”.

See: wc_Hash

Return:

- Success A positive return value indicates the digest size for the hash.
- Error Returns HASH_TYPE_E if hash_type is not supported.
- Failure Returns BAD_FUNC_ARG if an invalid hash_type was used.

Example

```
int hash_len = wc_HashGetDigestSize(hash_type);
if (hash_len <= 0) {
WOLFSSL_MSG("Invalid hash type/len");
return BAD_FUNC_ARG;
}
```

C.28.2.3 function wc_Hash

```
int wc_Hash(
    enum wc_HashType hash_type,
    const byte * data,
```

```
    word32 data_len,
    byte * hash,
    word32 hash_len
)
```

This function performs a hash on the provided data buffer and returns it in the hash buffer provided.

Parameters:

- **hash_type** A hash type from the "enum wc_HashType" such as "WC_HASH_TYPE_SHA256".
- **data** Pointer to buffer containing the data to hash.
- **data_len** Length of the data buffer.
- **hash** Pointer to buffer used to output the final hash to.
- **hash_len** Length of the hash buffer.

See: wc_HashGetDigestSize

Return: 0 Success, else error (such as BAD_FUNC_ARG or BUFFER_E).

Example

```
enum wc_HashType hash_type = WC_HASH_TYPE_SHA256;
int hash_len = wc_HashGetDigestSize(hash_type);
if (hash_len > 0) {
    int ret = wc_Hash(hash_type, data, data_len, hash_data, hash_len);
    if(ret == 0) {
        // Success
    }
}
```

C.28.2.4 function wc_HashUpdate

```
int wc_HashUpdate(
    wc_HashAlg * hash,
    enum wc_HashType type,
    const byte * data,
    word32 dataSz
)
```

This function updates the hash in a wc_HashAlg object with the provided data. The hash algorithm used is the one stored in the wc_HashAlg object when it was initialized with wc_HashInit() or wc_HashInit_ex().

Parameters:

- **hash** Pointer to the wc_HashAlg object, previously initialized with wc_HashInit() or wc_HashInit_ex().
- **type** Must be either WC_HASH_TYPE_NONE or the same hash type that was used to initialize the hash object. The hash type stored in the hash object during initialization is always used to select the algorithm. Any other value results in BAD_FUNC_ARG.
- **data** Pointer to buffer containing the data to hash. May be NULL only if dataSz is 0.
- **dataSz** Length of the data buffer in bytes.

See:

- wc_HashFinal
- wc_HashFree

Return:

- 0 Success.

- BAD_FUNC_ARG hash is NULL, data is NULL while dataSz is greater than 0, the hash object was not initialized with a valid hash type, or type is not WC_HASH_TYPE_NONE and does not match the hash type used during initialization.
- HASH_TYPE_E the hash type stored in the hash object is not supported or not compiled in.
- Other negative error codes may be returned by the underlying hash algorithm's update function.

Example

```
wc_HashAlg hash;
byte digest[WC_SHA256_DIGEST_SIZE];
byte data[] = { 0x61, 0x62, 0x63 };
int ret;

ret = wc_HashInit(&hash, WC_HASH_TYPE_SHA256);
if (ret == 0) {
    ret = wc_HashUpdate(&hash, WC_HASH_TYPE_SHA256, data, sizeof(data));
    if (ret == 0) {
        ret = wc_HashFinal(&hash, WC_HASH_TYPE_SHA256, digest);
    }
    wc_HashFree(&hash, WC_HASH_TYPE_SHA256);
}
```

C.28.2.5 function wc_HashFinal

```
int wc_HashFinal(
    wc_HashAlg * hash,
    enum wc_HashType type,
    byte * out
)
```

This function finalizes the hash in a wc_HashAlg object and writes the resulting digest to out. The hash algorithm used is the one stored in the wc_HashAlg object when it was initialized with wc_HashInit() or wc_HashInit_ex().

Parameters:

- **hash** Pointer to the wc_HashAlg object, previously initialized with wc_HashInit() or wc_HashInit_ex().
- **type** Must be either WC_HASH_TYPE_NONE or the same hash type that was used to initialize the hash object. The hash type stored in the hash object during initialization is always used to select the algorithm. Any other value results in BAD_FUNC_ARG.
- **out** Pointer to the buffer to receive the digest. It must be at least as large as the digest size of the hash type the object was initialized with (see wc_HashGetDigestSize()).

See:

- wc_HashUpdate
- wc_HashFree
- wc_HashGetDigestSize

Return:

- 0 Success.
- BAD_FUNC_ARG hash or out is NULL, the hash object was not initialized with a valid hash type, or type is not WC_HASH_TYPE_NONE and does not match the hash type used during initialization.
- HASH_TYPE_E the hash type stored in the hash object is not supported or not compiled in.
- Other negative error codes may be returned by the underlying hash algorithm's final function.

Example

```
wc_HashAlg hash;
byte digest[WC_SHA256_DIGEST_SIZE];
byte data[] = { 0x61, 0x62, 0x63 };
int ret;

ret = wc_HashInit(&hash, WC_HASH_TYPE_SHA256);
if (ret == 0) {
    ret = wc_HashUpdate(&hash, WC_HASH_TYPE_SHA256, data, sizeof(data));
    if (ret == 0) {
        ret = wc_HashFinal(&hash, WC_HASH_TYPE_SHA256, digest);
    }
    wc_HashFree(&hash, WC_HASH_TYPE_SHA256);
}
```

C.28.2.6 function wc_HashFree

```
int wc_HashFree(
    wc_HashAlg * hash,
    enum wc_HashType type
)
```

This function frees any resources held by a wc_HashAlg object. The hash algorithm used is the one stored in the wc_HashAlg object when it was initialized with wc_HashInit() or wc_HashInit_ex(). The wc_HashAlg object itself is not freed.

Parameters:

- **hash** Pointer to the wc_HashAlg object, previously initialized with wc_HashInit() or wc_HashInit_ex().
- **type** Must be either WC_HASH_TYPE_NONE or the same hash type that was used to initialize the hash object. The hash type stored in the hash object during initialization is always used to select the algorithm. Any other value results in BAD_FUNC_ARG.

See:

- wc_HashUpdate
- wc_HashFinal

Return:

- 0 Success.
- BAD_FUNC_ARG hash is NULL, the hash object was not initialized with a valid hash type, or type is not WC_HASH_TYPE_NONE and does not match the hash type used during initialization.
- HASH_TYPE_E the hash type stored in the hash object is not supported or not compiled in.

Example

```
wc_HashAlg hash;

if (wc_HashInit(&hash, WC_HASH_TYPE_SHA256) == 0) {
    // use hash
    wc_HashFree(&hash, WC_HASH_TYPE_SHA256);
}
```

C.28.2.7 function wc_HashSetFlags

```
int wc_HashSetFlags(
    wc_HashAlg * hash,
    enum wc_HashType type,
```

```
    word32 flags
)
```

This function sets the flags of the underlying hash object in a wc_HashAlg object. The hash algorithm used is the one stored in the wc_HashAlg object when it was initialized with wc_HashInit() or wc_HashInit_ex(). Available when WOLFSSL_HASH_FLAGS is defined.

Parameters:

- **hash** Pointer to the wc_HashAlg object, previously initialized with wc_HashInit() or wc_HashInit_ex().
- **type** Must be either WC_HASH_TYPE_NONE or the same hash type that was used to initialize the hash object. The hash type stored in the hash object during initialization is always used to select the algorithm. Any other value results in BAD_FUNC_ARG.
- **flags** Flags to set (for example WC_HASH_FLAG_ISCOPY).

See: wc_HashGetFlags

Return:

- 0 Success.
- BAD_FUNC_ARG hash is NULL, the hash object was not initialized with a valid hash type, or type is not WC_HASH_TYPE_NONE and does not match the hash type used during initialization.
- HASH_TYPE_E the hash type stored in the hash object is not supported or not compiled in.

Example

```
wc_HashAlg hash;

if (wc_HashInit(&hash, WC_HASH_TYPE_SHA256) == 0) {
    wc_HashSetFlags(&hash, WC_HASH_TYPE_SHA256, WC_HASH_FLAG_NONE);
    wc_HashFree(&hash, WC_HASH_TYPE_SHA256);
}
```

C.28.2.8 function wc_HashGetFlags

```
int wc_HashGetFlags(
    wc_HashAlg * hash,
    enum wc_HashType type,
    word32 * flags
)
```

This function gets the flags of the underlying hash object in a wc_HashAlg object. The hash algorithm used is the one stored in the wc_HashAlg object when it was initialized with wc_HashInit() or wc_HashInit_ex(). Available when WOLFSSL_HASH_FLAGS is defined.

Parameters:

- **hash** Pointer to the wc_HashAlg object, previously initialized with wc_HashInit() or wc_HashInit_ex().
- **type** Must be either WC_HASH_TYPE_NONE or the same hash type that was used to initialize the hash object. The hash type stored in the hash object during initialization is always used to select the algorithm. Any other value results in BAD_FUNC_ARG.
- **flags** Pointer to receive the current flags. If NULL, no flags are returned.

See: wc_HashSetFlags

Return:

- 0 Success.
- BAD_FUNC_ARG hash is NULL, the hash object was not initialized with a valid hash type, or type is not WC_HASH_TYPE_NONE and does not match the hash type used during initialization.
- HASH_TYPE_E the hash type stored in the hash object is not supported or not compiled in.

Example

```
wc_HashAlg hash;
word32 flags = 0;

if (wc_HashInit(&hash, WC_HASH_TYPE_SHA256) == 0) {
    wc_HashGetFlags(&hash, WC_HASH_TYPE_SHA256, &flags);
    wc_HashFree(&hash, WC_HASH_TYPE_SHA256);
}
```

C.28.2.9 function wc_Md5Hash

```
int wc_Md5Hash(
    const byte * data,
    word32 len,
    byte * hash
)
```

Convenience function, handles all the hashing and places the result into hash.

Parameters:

- **data** the data to hash
- **len** the length of data
- **hash** Byte array to hold hash value.
- **hashLen** Number of bytes to write to hash.

See:

- wc_Md5Hash
- wc_Md5Final
- wc_InitMd5

Return:

- 0 Returned upon successfully hashing the data.
- Memory_E memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Example

```
const byte* data;
word32 data_len;
byte* hash;
int ret;
...
ret = wc_Md5Hash(data, data_len, hash);
if (ret != 0) {
     // Md5 Hash Failure Case.
}
```

C.28.2.10 function wc_ShaHash

```
int wc_ShaHash(
    const byte * data,
    word32 len,
    byte * hash
)
```

Convenience function, handles all the hashing and places the result into hash.

Parameters:

- **data** the data to hash
- **len** the length of data
- **hash** Byte array to hold hash value.

See:

- wc_ShaHash
- wc_ShaFinal
- wc_InitSha

Return:

- 0 Returned upon successfully ....
- Memory_E memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Example

none

C.28.2.11 function wc_Sha224Hash

```
int wc_Sha224Hash(
    const byte * data,
    word32 len,
    byte * hash
)
```

Convenience function, handles all the hashing and places the result into hash.

Parameters:

- **data** the data to hash
- **len** the length of data
- **hash** Byte array to hold hash value.

See:

- wc_InitSha224
- wc_Sha224Update
- wc_Sha224Final

Return:

- 0 Success
- <0 Error

Example

none

C.28.2.12 function wc_Sha256Hash

```
int wc_Sha256Hash(
    const byte * data,
    word32 len,
    byte * hash
)
```

Convenience function, handles all the hashing and places the result into hash.

Parameters:

- **data** the data to hash
- **len** the length of data
- **hash** Byte array to hold hash value.

See:

- wc_Sha256Hash
- wc_Sha256Final
- wc_InitSha256

Return:

- 0 Returned upon successfully ...
- Memory_E memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Example

none

C.28.2.13 function wc_Sha384Hash

```
int wc_Sha384Hash(
    const byte * data,
    word32 len,
    byte * hash
)
```

Convenience function, handles all the hashing and places the result into hash.

Parameters:

- **data** the data to hash
- **len** the length of data
- **hash** Byte array to hold hash value.

See:

- wc_Sha384Hash
- wc_Sha384Final
- wc_InitSha384

Return:

- 0 Returned upon successfully hashing the data
- Memory_E memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Example

none

C.28.2.14 function wc_Sha512Hash

```
int wc_Sha512Hash(
    const byte * data,
    word32 len,
    byte * hash
)
```

Convenience function, handles all the hashing and places the result into hash.

Parameters:

- **data** the data to hash
- **len** the length of data
- **hash** Byte array to hold hash value.

See:

- wc_Sha512Hash
- wc_Sha512Final
- wc_InitSha512

Return:

- 0 Returned upon successfully hashing the inputted data
- Memory_E memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Example

none

C.28.2.15 function wc_Sha3_224Hash

```
int wc_Sha3_224Hash(
    const byte * data,
    word32 len,
    byte * hash
)
```

Convenience function, handles all the hashing and places the result into hash.

Parameters:

- **data** the data to hash
- **len** the length of data
- **hash** Byte array to hold hash value.

See:

- wc_InitSha3_224
- wc_Sha3_224_Update
- wc_Sha3_224_Final

Return:

- 0 Returned upon successfully hashing the data
- Memory_E memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Example

none

C.28.2.16 function wc_Sha3_256Hash

```
int wc_Sha3_256Hash(
    const byte * data,
    word32 len,
    byte * hash
)
```

Convenience function, handles all the hashing and places the result into hash.

Parameters:

- **data** the data to hash
- **len** the length of data
- **hash** Byte array to hold hash value.

See:

- wc_InitSha3_256
- wc_Sha3_256_Update
- wc_Sha3_256_Final

Return:

- 0 Returned upon successfully hashing the data
- Memory_E memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Example

none

C.28.2.17 function wc_Sha3_384Hash

```
int wc_Sha3_384Hash(
    const byte * data,
    word32 len,
    byte * hash
)
```

Convenience function, handles all the hashing and places the result into hash.

Parameters:

- **data** the data to hash
- **len** the length of data
- **hash** Byte array to hold hash value.

See:

- wc_InitSha3_384
- wc_Sha3_384_Update
- wc_Sha3_384_Final

Return:

- 0 Returned upon successfully hashing the data
- Memory_E memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Example

none

C.28.2.18 function wc_Sha3_512Hash

```
int wc_Sha3_512Hash(
    const byte * data,
    word32 len,
    byte * hash
)
```

Convenience function, handles all the hashing and places the result into hash.

Parameters:

- **data** the data to hash
- **len** the length of data
- **hash** Byte array to hold hash value.

See:

- wc_InitSha3_512
- wc_Sha3_512_Update
- wc_Sha3_512_Final

Return:

- 0 Returned upon successfully hashing the inputted data
- Memory_E memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Example

none

C.28.2.19 function wc_Shake128Hash

```
int wc_Shake128Hash(
    const byte * data,
    word32 len,
    byte * hash,
    word32 hashLen
)
```

Convenience function, handles all the hashing and places the result into hash.

Parameters:

- **data** the data to hash
- **len** the length of data
- **hash** Byte array to hold hash value.

See:

- wc_InitShake128
- wc_Shake128_Update
- wc_Shake128_Final

Return:

- 0 Returned upon successfully hashing the inputted data
- Memory_E memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Example

none

C.28.2.20 function wc_Shake256Hash

```
int wc_Shake256Hash(
    const byte * data,
    word32 len,
    byte * hash,
```

```
    word32 hashLen
)
```

Convenience function, handles all the hashing and places the result into hash.

Parameters:

- **data** the data to hash
- **len** the length of data
- **hash** Byte array to hold hash value.
- **hashLen** Number of bytes to write to hash.

See:

- wc_InitShake256
- wc_Shake256_Update
- wc_Shake256_Final

Return:

- 0 Returned upon successfully hashing the inputted data
- Memory_E memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Example

none

C.28.3 Source code

```
int wc_HashGetOID(enum wc_HashType hash_type);

int wc_HashGetDigestSize(enum wc_HashType hash_type);

int wc_Hash(enum wc_HashType hash_type,
    const byte* data, word32 data_len,
    byte* hash, word32 hash_len);

int wc_HashUpdate(wc_HashAlg* hash, enum wc_HashType type, const byte* data,
    word32 dataSz);

int wc_HashFinal(wc_HashAlg* hash, enum wc_HashType type, byte* out);

int wc_HashFree(wc_HashAlg* hash, enum wc_HashType type);

int wc_HashSetFlags(wc_HashAlg* hash, enum wc_HashType type, word32 flags);

int wc_HashGetFlags(wc_HashAlg* hash, enum wc_HashType type, word32* flags);

int wc_Md5Hash(const byte* data, word32 len, byte* hash);

int wc_ShaHash(const byte* data, word32 len, byte* hash);

int wc_Sha224Hash(const byte* data, word32 len, byte* hash);

int wc_Sha256Hash(const byte* data, word32 len, byte* hash);
```

```
int wc_Sha384Hash(const byte* data, word32 len, byte* hash);

int wc_Sha512Hash(const byte* data, word32 len, byte* hash);

int wc_Sha3_224Hash(const byte* data, word32 len, byte* hash);

int wc_Sha3_256Hash(const byte* data, word32 len, byte* hash);

int wc_Sha3_384Hash(const byte* data, word32 len, byte* hash);

int wc_Sha3_512Hash(const byte* data, word32 len, byte* hash);

int wc_Shake128Hash(const byte* data, word32 len, byte* hash,
                    word32 hashLen);

int wc_Shake256Hash(const byte* data, word32 len, byte* hash,
                    word32 hashLen);
```

C.29 hmac.h

C.29.1 Functions

	Name
int	**wc_HmacSetKey**(Hmac * hmac, int type, const byte * key, word32 keySz)This function initializes an Hmac object, setting its encryption type, key and HMAC length.
int	**wc_HmacUpdate**(Hmac * hmac, const byte * in, word32 sz)This function updates the message to authenticate using HMAC. It should be called after the Hmac object has been initialized with wc_HmacSetKey. This function may be called multiple times to update the message to hash. After calling wc_HmacUpdate as desired, one should call wc_HmacFinal to obtain the final authenticated message tag.
int	**wc_HmacFinal**(Hmac * hmac, byte * out)This function computes the final hash of an Hmac object's message.
int	**wolfSSL_GetHmacMaxSize**(void)This function returns the largest HMAC digest size available based on the configured cipher suites.
int	**wc_HKDF**(int type, const byte * inKey, word32 inKeySz, const byte * salt, word32 saltSz, const byte * info, word32 infoSz, byte * out, word32 outSz)This function provides access to a HMAC Key Derivation Function (HKDF). It utilizes HMAC to convert inKey, with an optional salt and optional info into a derived key, which it stores in out. The hash type defaults to MD5 if 0 or NULL is given.

	Name
int	**wc_HKDF_Extract**(int type, const byte * salt, word32 saltSz, const byte * inKey, word32 inKeySz, byte * out)This function provides access to a HMAC Key Derivation Function (HKDF). It utilizes HMAC to convert inKey, with an optional salt into a derived key, which it stores in out. The hash type defaults to MD5 if 0 or NULL is given.
int	**wc_HKDF_Extract_ex**(int type, const byte * salt, word32 saltSz, const byte * inKey, word32 inKeySz, byte * out, void * heap, int devId)This function provides access to a HMAC Key Derivation Function (HKDF). It utilizes HMAC to convert inKey, with an optional salt into a derived key, which it stores in out. The hash type defaults to MD5 if 0 or NULL is given. This is the _ex version adding heap hint and device identifier.
int	**wc_HKDF_Expand**(int type, const byte * inKey, word32 inKeySz, const byte * info, word32 infoSz, byte * out, word32 outSz)This function provides access to a HMAC Key Derivation Function (HKDF). It utilizes HMAC to convert inKey, with optional info into a derived key, which it stores in out. The hash type defaults to MD5 if 0 or NULL is given.
int	**wc_HKDF_Expand_ex**(int type, const byte * inKey, word32 inKeySz, const byte * info, word32 infoSz, byte * out, word32 outSz, void * heap, int devId)This function provides access to a HMAC Key Derivation Function (HKDF). It utilizes HMAC to convert inKey, with optional info into a derived key, which it stores in out. The hash type defaults to MD5 if 0 or NULL is given. This is the _ex version adding heap hint and device identifier.
int	**wc_Tls13_HKDF_Extract**(byte * prk, const byte * salt, word32 saltLen, byte * ikm, word32 ikmLen, int digest)This function provides access to RFC 5869 HMAC_based Extract_and_Expand Key Derivation Function (HKDF) for TLS v1.3 key derivation.
int	**wc_Tls13_HKDF_Extract_ex**(byte * prk, const byte * salt, word32 saltLen, byte * ikm, word32 ikmLen, int digest, void * heap, int devId)This function provides access to RFC 5869 HMAC_based Extract_and_Expand Key Derivation Function (HKDF) for TLS v1.3 key derivation. This is the _ex version adding heap hint and device identifier.

	Name
int	**wc_Tls13_HKDF_Expand_Label_ex**(byte * okm, word32 okmLen, const byte * prk, word32 prkLen, const byte * protocol, word32 protocolLen, const byte * label, word32 labelLen, const byte * info, word32 infoLen, int digest, void * heap, int devId)Expand data using HMAC, salt and label and info. TLS v1.3 defines this function for key derivation. This is the _ex version adding heap hint and device identifier.
int	**wc_Tls13_HKDF_Expand_Label**(byte * okm, word32 okmLen, const byte * prk, word32 prkLen, const byte * protocol, word32 protocolLen, const byte * label, word32 labelLen, const byte * info, word32 infoLen, int digest)Expand data using HMAC, salt and label and info. TLS v1.3 defines this function for key derivation.
int	**wc_Tls13_HKDF_Expand_Label_Alloc**(byte * okm, word32 okmLen, const byte * prk, word32 prkLen, const byte * protocol, word32 protocolLen, const byte * label, word32 labelLen, const byte * info, word32 infoLen, int digest, void * heap)This functions is very similar to wc_Tls13_HKDF_Expand_Label(), but it allocates memory if the stack space usually used isn't enough. Expand data using HMAC, salt and label and info. TLS v1.3 defines this function for key derivation. This is the _ex version adding heap hint and device identifier.

C.29.2 Functions Documentation

C.29.2.1 function wc_HmacSetKey

```
int wc_HmacSetKey(
    Hmac * hmac,
    int type,
    const byte * key,
    word32 keySz
)
```

This function initializes an Hmac object, setting its encryption type, key and HMAC length.

Parameters:

- **hmac** pointer to the Hmac object to initialize
- **type** type specifying which encryption method the Hmac object should use. Valid options are: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512
- **key** pointer to a buffer containing the key with which to initialize the Hmac object
- **length** length of the key

See:

- wc_HmacUpdate
- wc_HmacFinal

Return:

- 0 Returned on successfully initializing the Hmac object
- BAD_FUNC_ARG Returned if the input type is invalid (see type param)
- MEMORY_E Returned if there is an error allocating memory for the structure to use for hashing
- HMAC_MIN_KEYLEN_E Returned when using a FIPS implementation and the key length specified is shorter than the minimum acceptable FIPS standard of 14 bytes

Example

```
Hmac hmac;
byte key[] = { // initialize with key to use for encryption };
if (wc_HmacSetKey(&hmac, WC_MD5, key, sizeof(key)) != 0) {
    // error initializing Hmac object
}
```

C.29.2.2 function wc_HmacUpdate

```
int wc_HmacUpdate(
    Hmac * hmac,
    const byte * in,
    word32 sz
)
```

This function updates the message to authenticate using HMAC. It should be called after the Hmac object has been initialized with wc_HmacSetKey. This function may be called multiple times to update the message to hash. After calling wc_HmacUpdate as desired, one should call wc_HmacFinal to obtain the final authenticated message tag.

Parameters:

- **hmac** pointer to the Hmac object for which to update the message
- **msg** pointer to the buffer containing the message to append
- **length** length of the message to append

See:

- wc_HmacSetKey
- wc_HmacFinal

Return:

- 0 Returned on successfully updating the message to authenticate
- MEMORY_E Returned if there is an error allocating memory for use with a hashing algorithm

Example

```
Hmac hmac;
byte msg[] = { // initialize with message to authenticate };
byte msg2[] = { // initialize with second half of message };
// initialize hmac
if( wc_HmacUpdate(&hmac, msg, sizeof(msg)) != 0) {
    // error updating message
}
if( wc_HmacUpdate(&hmac, msg2, sizeof(msg)) != 0) {
    // error updating with second message
}
```

C.29.2.3 function wc_HmacFinal

```
int wc_HmacFinal(
    Hmac * hmac,
    byte * out
)
```

This function computes the final hash of an Hmac object's message.

Parameters:

- **hmac** pointer to the Hmac object for which to calculate the final hash
- **hash** pointer to the buffer in which to store the final hash. Should have room available as required by the hashing algorithm chosen

See:

- wc_HmacSetKey
- wc_HmacUpdate

Return:

- 0 Returned on successfully computing the final hash
- MEMORY_E Returned if there is an error allocating memory for use with a hashing algorithm

Example

```
Hmac hmac;
byte hash[MD5_DIGEST_SIZE];
// initialize hmac with MD5 as type
// wc_HmacUpdate() with messages

if (wc_HmacFinal(&hmac, hash) != 0) {
    // error computing hash
}
```

C.29.2.4 function wolfSSL_GetHmacMaxSize

```
int wolfSSL_GetHmacMaxSize(
    void
)
```

This function returns the largest HMAC digest size available based on the configured cipher suites.

Parameters:

- **none** No parameters.

See: none

Return: Success Returns the largest HMAC digest size available based on the configured cipher suites

Example

```
int maxDigestSz = wolfSSL_GetHmacMaxSize();
```

C.29.2.5 function wc_HKDF

```
int wc_HKDF(
    int type,
    const byte * inKey,
    word32 inKeySz,
    const byte * salt,
```

```
    word32 saltSz,
    const byte * info,
    word32 infoSz,
    byte * out,
    word32 outSz
)
```

This function provides access to a HMAC Key Derivation Function (HKDF). It utilizes HMAC to convert inKey, with an optional salt and optional info into a derived key, which it stores in out. The hash type defaults to MD5 if 0 or NULL is given.

Parameters:

- **type** hash type to use for the HKDF. Valid types are: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512
- **inKey** pointer to the buffer containing the key to use for KDF
- **inKeySz** length of the input key
- **salt** pointer to a buffer containing an optional salt. Use NULL instead if not using a salt
- **saltSz** length of the salt. Use 0 if not using a salt
- **info** pointer to a buffer containing optional additional info. Use NULL if not appending extra info
- **infoSz** length of additional info. Use 0 if not using additional info
- **out** pointer to the buffer in which to store the derived key
- **outSz** space available in the output buffer to store the generated key

See: wc_HmacSetKey

Return:

- 0 Returned upon successfully generating a key with the given inputs
- BAD_FUNC_ARG Returned if an invalid hash type is given (see type param)
- MEMORY_E Returned if there is an error allocating memory
- HMAC_MIN_KEYLEN_E May be returned when using a FIPS implementation and the key length specified is shorter than the minimum acceptable FIPS standard

The HMAC configure option is –enable-hmac (on by default) or if building sources directly HAVE_HKDF

Example

```
byte key[] = { // initialize with key };
byte salt[] = { // initialize with salt };
byte derivedKey[MAX_DIGEST_SIZE];

int ret = wc_HKDF(WC_SHA512, key, sizeof(key), salt, sizeof(salt),
NULL, 0, derivedKey, sizeof(derivedKey));
if ( ret != 0 ) {
    // error generating derived key
}
```

C.29.2.6 function wc_HKDF_Extract

```
int wc_HKDF_Extract(
    int type,
    const byte * salt,
    word32 saltSz,
    const byte * inKey,
    word32 inKeySz,
    byte * out
)
```

This function provides access to a HMAC Key Derivation Function (HKDF). It utilizes HMAC to convert inKey, with an optional salt into a derived key, which it stores in out. The hash type defaults to MD5 if 0 or NULL is given.

Parameters:

- **type** hash type to use for the HKDF. Valid types are: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512
- **salt** pointer to a buffer containing an optional salt. Use NULL instead if not using a salt
- **saltSz** length of the salt. Use 0 if not using a salt
- **inKey** pointer to the buffer containing the key to use for KDF
- **inKeySz** length of the input key
- **out** pointer to the buffer in which to store the derived key

See:

- wc_HKDF
- wc_HKDF_Extract_ex
- wc_HKDF_Expand
- wc_HKDF_Expand_ex

Return:

- 0 Returned upon successfully generating a key with the given inputs
- BAD_FUNC_ARG Returned if an invalid hash type is given (see type param)
- MEMORY_E Returned if there is an error allocating memory
- HMAC_MIN_KEYLEN_E May be returned when using a FIPS implementation and the key length specified is shorter than the minimum acceptable FIPS standard

The HMAC configure option is –enable-hmac (on by default) or if building sources directly HAVE_HKDF

Example

```
byte key[] = { // initialize with key };
byte salt[] = { // initialize with salt };
byte derivedKey[MAX_DIGEST_SIZE];

int ret = wc_HKDF_Extract(WC_SHA512, salt, sizeof(salt), key, sizeof(key),
    derivedKey);
if ( ret != 0 ) {
    // error generating derived key
}
```

C.29.2.7 function wc_HKDF_Extract_ex

```
int wc_HKDF_Extract_ex(
    int type,
    const byte * salt,
    word32 saltSz,
    const byte * inKey,
    word32 inKeySz,
    byte * out,
    void * heap,
    int devId
)
```

This function provides access to a HMAC Key Derivation Function (HKDF). It utilizes HMAC to convert inKey, with an optional salt into a derived key, which it stores in out. The hash type defaults to MD5 if 0 or NULL is given. This is the _ex version adding heap hint and device identifier.

Parameters:

- **type** hash type to use for the HKDF. Valid types are: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512
- **salt** pointer to a buffer containing an optional salt. Use NULL instead if not using a salt
- **saltSz** length of the salt. Use 0 if not using a salt
- **inKey** pointer to the buffer containing the key to use for KDF
- **inKeySz** length of the input key
- **out** pointer to the buffer in which to store the derived key. Must be digest length per hash 'type'. See wc_HmacSizeByType()
- **heap** heap hint to use for memory. Can be NULL
- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID (-2) if not used

See:

- wc_HKDF
- wc_HKDF_Extract
- wc_HKDF_Expand
- wc_HKDF_Expand_ex
- wc_HmacSizeByType

Return:

- 0 Returned upon successfully generating a key with the given inputs
- BAD_FUNC_ARG Returned if an invalid hash type is given (see type param)
- MEMORY_E Returned if there is an error allocating memory
- HMAC_MIN_KEYLEN_E May be returned when using a FIPS implementation and the key length specified is shorter than the minimum acceptable FIPS standard
- WC_PENDING_E May be returned in a WOLF_CRYPTO_CB build when the registered crypto callback device has taken the request but not yet finished it. The caller must re-invoke with identical arguments until the result is no longer WC_PENDING_E; HKDF has no WC_ASYNC_DEV, so this is a poll and not a wc_AsyncWait(). In a WOLFSSL_ASYNC_CRYPT build the TLS 1.3 key schedule resumes a pending HKDF request by re-invoking the callback with identical arguments; without WOLFSSL_ASYNC_CRYPT a device also used for TLS 1.3 must complete HKDF requests synchronously. wc_HKDF_ex() follows the same contract and re-issues its extract step on every retry, so a device that pends must serve a repeated identical request from its completed result. There is no request handle: a device should key its completion tracking on the output pointer plus the argument tuple, and one that cannot correlate a retry that way must not pend.

The HMAC configure option is –enable-hmac (on by default) or if building sources directly HAVE_HKDF

Example

```
byte key[] = { // initialize with key };
byte salt[] = { // initialize with salt };
byte derivedKey[MAX_DIGEST_SIZE];

int ret = wc_HKDF_Extract_ex(WC_SHA512, salt, sizeof(salt), key, sizeof(key),
    derivedKey, NULL, INVALID_DEVID);
if ( ret != 0 ) {
    // error generating derived key
}
```

C.29.2.8 function wc_HKDF_Expand

```
int wc_HKDF_Expand(
    int type,
    const byte * inKey,
```

```
    word32 inKeySz,
    const byte * info,
    word32 infoSz,
    byte * out,
    word32 outSz
)
```

This function provides access to a HMAC Key Derivation Function (HKDF). It utilizes HMAC to convert inKey, with optional info into a derived key, which it stores in out. The hash type defaults to MD5 if 0 or NULL is given.

Parameters:

- **type** hash type to use for the HKDF. Valid types are: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512
- **inKey** pointer to the buffer containing the key to use for KDF
- **inKeySz** length of the input key
- **info** pointer to a buffer containing optional additional info. Use NULL if not appending extra info
- **infoSz** length of additional info. Use 0 if not using additional info
- **out** pointer to the buffer in which to store the derived key
- **outSz** space available in the output buffer to store the generated key

See:

- wc_HKDF
- wc_HKDF_Extract
- wc_HKDF_Extract_ex
- wc_HKDF_Expand_ex

Return:

- 0 Returned upon successfully generating a key with the given inputs
- BAD_FUNC_ARG Returned if an invalid hash type is given (see type param)
- MEMORY_E Returned if there is an error allocating memory
- HMAC_MIN_KEYLEN_E May be returned when using a FIPS implementation and the key length specified is shorter than the minimum acceptable FIPS standard

The HMAC configure option is –enable-hmac (on by default) or if building sources directly HAVE_HKDF

Example

```
byte key[] = { // initialize with key };
byte salt[] = { // initialize with salt };
byte derivedKey[MAX_DIGEST_SIZE];

int ret = wc_HKDF_Expand(WC_SHA512, key, sizeof(key), NULL, 0,
    derivedKey, sizeof(derivedKey));
if ( ret != 0 ) {
    // error generating derived key
}
```

C.29.2.9 function wc_HKDF_Expand_ex

```
int wc_HKDF_Expand_ex(
    int type,
    const byte * inKey,
    word32 inKeySz,
    const byte * info,
    word32 infoSz,
```

```
    byte * out,
    word32 outSz,
    void * heap,
    int devId
)
```

This function provides access to a HMAC Key Derivation Function (HKDF). It utilizes HMAC to convert inKey, with optional info into a derived key, which it stores in out. The hash type defaults to MD5 if 0 or NULL is given. This is the _ex version adding heap hint and device identifier.

Parameters:

- **type** hash type to use for the HKDF. Valid types are: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512
- **inKey** pointer to the buffer containing the key to use for KDF
- **inKeySz** length of the input key
- **info** pointer to a buffer containing optional additional info. Use NULL if not appending extra info
- **infoSz** length of additional info. Use 0 if not using additional info
- **out** pointer to the buffer in which to store the derived key
- **outSz** space available in the output buffer to store the generated key
- **heap** heap hint to use for memory. Can be NULL
- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID (-2) if not used

See:

- wc_HKDF
- wc_HKDF_Extract
- wc_HKDF_Extract_ex
- wc_HKDF_Expand

Return:

- 0 Returned upon successfully generating a key with the given inputs
- BAD_FUNC_ARG Returned if an invalid hash type is given (see type param)
- MEMORY_E Returned if there is an error allocating memory
- HMAC_MIN_KEYLEN_E May be returned when using a FIPS implementation and the key length specified is shorter than the minimum acceptable FIPS standard
- WC_PENDING_E May be returned in a WOLF_CRYPTO_CB build when the registered crypto callback device has taken the request but not yet finished it. The caller must re-invoke with identical arguments until the result is no longer WC_PENDING_E; HKDF has no WC_ASYNC_DEV, so this is a poll and not a wc_AsyncWait(). In a WOLFSSL_ASYNC_CRYPT build the TLS 1.3 key schedule resumes a pending HKDF request by re-invoking the callback with identical arguments; without WOLFSSL_ASYNC_CRYPT a device also used for TLS 1.3 must complete HKDF requests synchronously. wc_HKDF_ex() follows the same contract and re-issues its extract step on every retry, so a device that pends must serve a repeated identical request from its completed result. There is no request handle: a device should key its completion tracking on the output pointer plus the argument tuple, and one that cannot correlate a retry that way must not pend.

The HMAC configure option is –enable-hmac (on by default) or if building sources directly HAVE_HKDF

Example

```
byte key[] = { // initialize with key };
byte salt[] = { // initialize with salt };
byte derivedKey[MAX_DIGEST_SIZE];

int ret = wc_HKDF_Expand_ex(WC_SHA512, key, sizeof(key), NULL, 0,
    derivedKey, sizeof(derivedKey), NULL, INVALID_DEVID);
if ( ret != 0 ) {
```

```
    // error generating derived key
}
```

C.29.2.10 function wc_Tls13_HKDF_Extract

```
int wc_Tls13_HKDF_Extract(
    byte * prk,
    const byte * salt,
    word32 saltLen,
    byte * ikm,
    word32 ikmLen,
    int digest
)
```

This function provides access to RFC 5869 HMAC-based Extract-and-Expand Key Derivation Function (HKDF) for TLS v1.3 key derivation.

Parameters:

- **prk** Generated pseudorandom key
- **salt** salt. May be NULL, in which case saltLen is ignored
- **saltLen** length of the salt
- **ikm** input keying material. May be NULL when ikmLen is 0
- **ikmLen** length of the input keying material. 0 substitutes a digest length zeroed IKM per RFC 8446, not an empty IKM
- **digest** hash type to use for the HKDF. Valid types are: WC_SHA256, WC_SHA384, WC_SHA512 or WC_SM3

See:

- wc_HKDF
- wc_HKDF_Extract
- wc_HKDF_Extract_ex
- wc_HKDF_Expand
- wc_Tls13_HKDF_Extract_ex

Return:

- 0 Returned upon successfully generating a key with the given inputs
- BAD_FUNC_ARG Returned if an invalid hash type is given (see digest param), if prk is NULL, or if ikm is NULL and ikmLen is not 0
- MEMORY_E Returned if there is an error allocating memory
- HMAC_MIN_KEYLEN_E May be returned when using a FIPS implementation and the key length specified is shorter than the minimum acceptable FIPS standard

Example

```
byte ikm[] = { // initialize with input keying material };
byte salt[] = { // initialize with optional salt };
byte prk[MAX_DIGEST_SIZE];

int ret = wc_Tls13_HKDF_Extract(prk, salt, sizeof(salt), ikm, sizeof(ikm),
    WC_SHA512);
if ( ret != 0 ) {
    // error generating derived key
}
```

C.29.2.11 function wc_Tls13_HKDF_Extract_ex

```
int wc_Tls13_HKDF_Extract_ex(
    byte * prk,
    const byte * salt,
    word32 saltLen,
    byte * ikm,
    word32 ikmLen,
    int digest,
    void * heap,
    int devId
)
```

This function provides access to RFC 5869 HMAC-based Extract-and-Expand Key Derivation Function (HKDF) for TLS v1.3 key derivation. This is the _ex version adding heap hint and device identifier.

Parameters:

- **prk** Generated pseudorandom key
- **salt** Salt. May be NULL; saltLen is then ignored unless a crypto callback handles the request
- **saltLen** Length of the salt
- **ikm** Input keying material. May be NULL when ikmLen is 0
- **ikmLen** Length of the input keying material. 0 substitutes a digest length zeroed IKM per RFC 8446, not an empty IKM
- **digest** Hash type to use for the HKDF. Valid types are: WC_SHA256, WC_SHA384, WC_SHA512 or WC_SM3
- **heap** Heap hint to use for memory. Can be NULL
- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID (-2) if not used

See:

- wc_HKDF
- wc_HKDF_Extract
- wc_HKDF_Extract_ex
- wc_HKDF_Expand
- wc_Tls13_HKDF_Extract

Return:

- 0 Returned upon successfully generating a key with the given inputs
- BAD_FUNC_ARG Returned if an invalid hash type is given (see digest param), if prk is NULL, or if ikm is NULL and ikmLen is not 0
- MEMORY_E Returned if there is an error allocating memory
- HMAC_MIN_KEYLEN_E May be returned when using a FIPS implementation and the key length specified is shorter than the minimum acceptable FIPS standard
- WC_PENDING_E May be returned in a WOLF_CRYPTO_CB build when the registered crypto callback device has taken the request but not yet finished it; the caller re-invokes with identical arguments until the result is no longer WC_PENDING_E. The TLS 1.3 key schedule does this in WOLFSSL_ASYNC_CRYPT builds.

Example

```
byte ikm[] = { // initialize with input keying material };
byte salt[] = { // initialize with optional salt };
byte prk[MAX_DIGEST_SIZE];

int ret = wc_Tls13_HKDF_Extract_ex(prk, salt, sizeof(salt), ikm,
    sizeof(ikm), WC_SHA512, NULL, INVALID_DEVID);
if ( ret != 0 ) {
```

```
    // error generating derived key
}
```

C.29.2.12 function wc_Tls13_HKDF_Expand_Label_ex

```
int wc_Tls13_HKDF_Expand_Label_ex(
    byte * okm,
    word32 okmLen,
    const byte * prk,
    word32 prkLen,
    const byte * protocol,
    word32 protocolLen,
    const byte * label,
    word32 labelLen,
    const byte * info,
    word32 infoLen,
    int digest,
    void * heap,
    int devId
)
```

Expand data using HMAC, salt and label and info. TLS v1.3 defines this function for key derivation. This is the _ex version adding heap hint and device identifier.

Parameters:

- **okm** Generated pseudorandom key - output key material.
- **okmLen** Length of generated pseudorandom key - output key material.
- **prk** Salt - pseudo-random key.
- **prkLen** Length of the salt - pseudo-random key.
- **protocol** TLS protocol label.
- **protocolLen** Length of the TLS protocol label.
- **info** Information to expand.
- **infoLen** Length of the information.
- **digest** Hash type to use for the HKDF. Valid types are: WC_SHA256, WC_SHA384 or WC_SHA512
- **heap** Heap hint to use for memory. Can be NULL
- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID (-2) if not used

See:

- wc_HKDF
- wc_HKDF_Extract
- wc_HKDF_Extract_ex
- wc_HKDF_Expand
- wc_Tls13_HKDF_Expand_Label
- wc_Tls13_HKDF_Expand_Label_Alloc

Return:

- 0 Returned upon successfully generating a key with the given inputs
- BAD_FUNC_ARG Returned if an invalid hash type is given (see type param)
- MEMORY_E Returned if there is an error allocating memory
- HMAC_MIN_KEYLEN_E May be returned when using a FIPS implementation and the key length specified is shorter than the minimum acceptable FIPS standard
- WC_PENDING_E May be returned in a WOLF_CRYPTO_CB build when the registered crypto callback device has taken the request but not yet finished it; the caller re-invokes with identical arguments until the result is no longer WC_PENDING_E. The TLS 1.3 key schedule does this in WOLFSSL_ASYNC_CRYPT builds.

C.29.2.13 function wc_Tls13_HKDF_Expand_Label

```
int wc_Tls13_HKDF_Expand_Label(
    byte * okm,
    word32 okmLen,
    const byte * prk,
    word32 prkLen,
    const byte * protocol,
    word32 protocolLen,
    const byte * label,
    word32 labelLen,
    const byte * info,
    word32 infoLen,
    int digest
)
```

Expand data using HMAC, salt and label and info. TLS v1.3 defines this function for key derivation.

Parameters:

- **okm** Generated pseudorandom key - output key material.
- **okmLen** Length of generated pseudorandom key - output key material.
- **prk** Salt - pseudo-random key.
- **prkLen** Length of the salt - pseudo-random key.
- **protocol** TLS protocol label.
- **protocolLen** Length of the TLS protocol label.
- **info** Information to expand.
- **infoLen** Length of the information.
- **digest** Hash type to use for the HKDF. Valid types are: WC_SHA256, WC_SHA384 or WC_SHA512

See:

- wc_HKDF
- wc_HKDF_Extract
- wc_HKDF_Extract_ex
- wc_HKDF_Expand
- wc_Tls13_HKDF_Expand_Label_ex
- wc_Tls13_HKDF_Expand_Label_Alloc

Return:

- 0 Returned upon successfully generating a key with the given inputs
- BAD_FUNC_ARG Returned if an invalid hash type is given (see type param)
- MEMORY_E Returned if there is an error allocating memory
- HMAC_MIN_KEYLEN_E May be returned when using a FIPS implementation and the key length specified is shorter than the minimum acceptable FIPS standard

C.29.2.14 function wc_Tls13_HKDF_Expand_Label_Alloc

```
int wc_Tls13_HKDF_Expand_Label_Alloc(
    byte * okm,
    word32 okmLen,
    const byte * prk,
    word32 prkLen,
    const byte * protocol,
    word32 protocolLen,
    const byte * label,
    word32 labelLen,
```

```
    const byte * info,
    word32 infoLen,
    int digest,
    void * heap
)
```

This functions is very similar to wc_Tls13_HKDF_Expand_Label(), but it allocates memory if the stack space usually used isn't enough. Expand data using HMAC, salt and label and info. TLS v1.3 defines this function for key derivation. This is the _ex version adding heap hint and device identifier.

Parameters:

- **okm** Generated pseudorandom key - output key material.
- **okmLen** Length of generated pseudorandom key - output key material.
- **prk** Salt - pseudo-random key.
- **prkLen** Length of the salt - pseudo-random key.
- **protocol** TLS protocol label.
- **protocolLen** Length of the TLS protocol label.
- **info** Information to expand.
- **infoLen** Length of the information.
- **digest** Hash type to use for the HKDF. Valid types are: WC_SHA256, WC_SHA384 or WC_SHA512
- **heap** Heap hint to use for memory. Can be NULL

See:

- wc_HKDF
- wc_HKDF_Extract
- wc_HKDF_Extract_ex
- wc_HKDF_Expand
- wc_Tls13_HKDF_Expand_Label
- wc_Tls13_HKDF_Expand_Label_ex

Return:

- 0 Returned upon successfully generating a key with the given inputs
- BAD_FUNC_ARG Returned if an invalid hash type is given (see type param)
- MEMORY_E Returned if there is an error allocating memory
- HMAC_MIN_KEYLEN_E May be returned when using a FIPS implementation and the key length specified is shorter than the minimum acceptable FIPS standard
- WC_PENDING_E May be returned in a WOLF_CRYPTO_CB build when the registered crypto callback device has taken the request but not yet finished it; the caller re-invokes with identical arguments until the result is no longer WC_PENDING_E. The TLS 1.3 key schedule does this in WOLFSSL_ASYNC_CRYPT builds.

C.29.3 Source code

```
int wc_HmacSetKey(Hmac* hmac, int type, const byte* key, word32 keySz);

int wc_HmacUpdate(Hmac* hmac, const byte* in, word32 sz);

int wc_HmacFinal(Hmac* hmac, byte* out);

int wolfSSL_GetHmacMaxSize(void);

int wc_HKDF(int type, const byte* inKey, word32 inKeySz,
                    const byte* salt, word32 saltSz,
```

```
                    const byte* info, word32 infoSz,
                    byte* out, word32 outSz);


int wc_HKDF_Extract(
    int type,
    const byte* salt, word32 saltSz,
    const byte* inKey, word32 inKeySz,
    byte* out);

int wc_HKDF_Extract_ex(
    int type,
    const byte* salt, word32 saltSz,
    const byte* inKey, word32 inKeySz,
    byte* out,
    void* heap, int devId);

int wc_HKDF_Expand(
    int type,
    const byte* inKey, word32 inKeySz,
    const byte* info, word32 infoSz,
    byte* out, word32 outSz);

int wc_HKDF_Expand_ex(
    int type,
    const byte* inKey, word32 inKeySz,
    const byte* info, word32 infoSz,
    byte* out, word32 outSz,
    void* heap, int devId);

int wc_Tls13_HKDF_Extract(
    byte* prk,
    const byte* salt, word32 saltLen,
    byte* ikm, word32 ikmLen, int digest);

int wc_Tls13_HKDF_Extract_ex(
    byte* prk,
    const byte* salt, word32 saltLen,
    byte* ikm, word32 ikmLen, int digest,
    void* heap, int devId);

int wc_Tls13_HKDF_Expand_Label_ex(
    byte* okm, word32 okmLen,
    const byte* prk, word32 prkLen,
    const byte* protocol, word32 protocolLen,
    const byte* label, word32 labelLen,
    const byte* info, word32 infoLen,
    int digest,
    void* heap, int devId);

int wc_Tls13_HKDF_Expand_Label(
    byte* okm, word32 okmLen,
    const byte* prk, word32 prkLen,
    const byte* protocol, word32 protocolLen,
```

```
    const byte* label, word32 labelLen,
    const byte* info, word32 infoLen,
    int digest);

int wc_Tls13_HKDF_Expand_Label_Alloc(
    byte* okm, word32 okmLen,
    const byte* prk, word32 prkLen,
    const byte* protocol, word32 protocolLen,
    const byte* label, word32 labelLen,
    const byte* info, word32 infoLen,
    int digest, void* heap);
```

C.30 iotsafe.h

C.30.1 Functions

	Name
int	**wolfSSL_CTX_iotsafe_enable**(WOLFSSL_CTX * ctx)This function enables the IoT-Safe support on the given context.
int	**wolfSSL_iotsafe_on**(WOLFSSL * ssl, byte privkey_id, byte ecdh_keypair_slot, byte peer_pubkey_slot, byte peer_cert_slot)This function connects the IoT-Safe TLS callbacks to the given SSL session.
int	**wolfSSL_iotsafe_on_ex**(WOLFSSL * ssl, byte * privkey_id, byte * ecdh_keypair_slot, byte * peer_pubkey_slot, byte * peer_cert_slot, word16 id_size)This function connects the IoT_Safe TLS callbacks to the given SSL session. This is equivalent to wolfSSL_iotsafe_on except that the IDs for the IoT-SAFE slots can be passed by reference, and the length of the ID fields can be specified via the parameter “id_size”.
void	**wolfIoTSafe_SetCSIM_read_cb**(wolfSSL_IOTSafe_CSIM_read_cb rf)Associates a read callback for the AT+CSIM commands. This input function is usually associated to a read event of a UART channel communicating with the modem. The read callback associated is global and changes for all the contexts that use IoT-safe support at the same time. This function must be called from a single thread, during initialization, before any IoT-Safe operation is performed.

	Name
void	**wolfIoTSafe_SetCSIM_write_cb**(wolfSSL_IOTSafe_CSIM_write_cb wf)Associates a write callback for the AT+CSIM commands. This output function is usually associated to a write event on a UART channel communicating with the modem. The write callback associated is global and changes for all the contexts that use IoT-safe support at the same time. This function must be called from a single thread, during initialization, before any IoT-Safe operation is performed.
int	**wolfIoTSafe_GetRandom**(unsigned char * out, word32 sz)Generate a random buffer of given size, using the IoT-Safe function GetRandom. This function is automatically used by the wolfCrypt RNG object.
int	**wolfIoTSafe_GetCert**(uint8_t id, unsigned char * output, unsigned long sz)Import a certificate stored in a file on IoT-Safe applet, and store it locally in memory. Works with one-byte file ID field.
int	**wolfIoTSafe_GetCert_ex**(uint8_t * id, uint16_t id_sz, unsigned char * output, unsigned long sz)Import a certificate stored in a file on IoT_Safe applet, and store it locally in memory. Equivalent to wolfIoTSafe_GetCert, except that it can be invoked with a file ID of two or more bytes.
int	**wc_iotsafe_ecc_import_public**(ecc_key * key, byte key_id)Import an ECC 256-bit public key, stored in the IoT-Safe applet, into an ecc_key object.
int	**wc_iotsafe_ecc_export_public**(ecc_key * key, byte key_id)Export an ECC 256-bit public key, from ecc_key object to a writable public-key slot into the IoT-Safe applet.
int	**wc_iotsafe_ecc_export_public_ex**(ecc_key * key, byte * key_id, word16 id_size)Export an ECC 256-bit public key, from ecc_key object to a writable public-key slot into the IoT-Safe applet. Equivalent to wc_iotsafe_ecc_export_public, except that it can be invoked with a key ID of two or more bytes.
int	**wc_iotsafe_ecc_import_public_ex**(ecc_key * key, byte * key_id, word16 id_size)Export an ECC 256_bit public key, from ecc_key object to a writable public_key slot into the IoT_Safe applet. Equivalent to wc_iotsafe_ecc_import_public, except that it can be invoked with a key ID of two or more bytes.

	Name
int	**wc_iotsafe_ecc_export_private**(ecc_key * key, byte key_id)Export an ECC 256-bit key, from ecc_key object to a writable private-key slot into the IoT-Safe applet.
int	**wc_iotsafe_ecc_export_private_ex**(ecc_key * key, byte * key_id, word16 id_size)Export an ECC 256_bit key, from ecc_key object to a writable private_key slot into the IoT_Safe applet. Equivalent to wc_iotsafe_ecc_export_private, except that it can be invoked with a key ID of two or more bytes.
int	**wc_iotsafe_ecc_sign_hash**(byte * in, word32 inlen, byte * out, word32 * outlen, byte key_id)Sign a pre-computed HASH, using a private key previously stored, or pre-provisioned, in the IoT-Safe applet.
int	**wc_iotsafe_ecc_sign_hash_ex**(byte * in, word32 inlen, byte * out, word32 * outlen, byte * key_id, word16 id_size)Sign a pre_computed HASH, using a private key previously stored, or pre_provisioned, in the IoT_Safe applet. Equivalent to wc_iotsafe_ecc_sign_hash, except that it can be invoked with a key ID of two or more bytes.
int	**wc_iotsafe_ecc_verify_hash**(byte * sig, word32 siglen, byte * hash, word32 hashlen, int * res, byte key_id)Verify an ECC signature against a pre-computed HASH, using a public key previously stored, or pre-provisioned, in the IoT-Safe applet. Result is written to res. 1 is valid, 0 is invalid. Note: Do not use the return value to test for valid. Only use res.
int	**wc_iotsafe_ecc_verify_hash_ex**(byte * sig, word32 siglen, byte * hash, word32 hashlen, int * res, byte * key_id, word16 id_size)Verify an ECC signature against a pre_computed HASH, using a public key previously stored, or pre_provisioned, in the IoT_Safe applet. Result is written to res. 1 is valid, 0 is invalid. Note: Do not use the return value to test for valid. Only use res. Equivalent to wc_iotsafe_ecc_verify_hash, except that it can be invoked with a key ID of two or more bytes.
int	**wc_iotsafe_ecc_gen_k**(byte key_id)Generate an ECC 256_bit keypair and store it in a (writable) slot into the IoT-Safe applet.
int	**wc_iotsafe_ecc_gen_k_ex**(byte * key_id, word16 id_size)Generate an ECC 256_bit keypair and store it in a (writable) slot into the IoT_Safe applet. Equivalent to wc_iotsafe_ecc_gen_k, except that it can be invoked with a key ID of two or more bytes.

C.30.2 Functions Documentation

C.30.2.1 function wolfSSL_CTX_iotsafe_enable

```
int wolfSSL_CTX_iotsafe_enable(
    WOLFSSL_CTX * ctx
)
```

This function enables the IoT-Safe support on the given context.

Parameters:

- **ctx** pointer to the WOLFSSL_CTX object on which the IoT-safe support must be enabled

See:

- wolfSSL_iotsafe_on
- wolfIoTSafe_SetCSIM_read_cb
- wolfIoTSafe_SetCSIM_write_cb

Return:

- 0 on success
- WC_HW_E on hardware error

This function loads the IoT-Safe applet on first use. In multi-threaded builds both the applet load and every individual APDU transaction are serialized internally on a port-level lock, so concurrent TLS sessions can safely share the same IoT-Safe context, and this function may be called concurrently for several contexts. The CSIM read and write callbacks must still be installed beforehand, from a single thread; see wolfIoTSafe_SetCSIM_read_cb() and wolfIoTSafe_SetCSIM_write_cb().

Example

```
WOLFSSL_CTX *ctx;
ctx = wolfSSL_CTX_new(wolfTLSv1_2_client_method());
if (!ctx)
    return NULL;
wolfSSL_CTX_iotsafe_enable(ctx);
```

C.30.2.2 function wolfSSL_iotsafe_on

```
int wolfSSL_iotsafe_on(
    WOLFSSL * ssl,
    byte privkey_id,
    byte ecdh_keypair_slot,
    byte peer_pubkey_slot,
    byte peer_cert_slot
)
```

This function connects the IoT-Safe TLS callbacks to the given SSL session.

Parameters:

- **ssl** pointer to the WOLFSSL object where the callbacks will be enabled
- **privkey_id** id of the iot-safe applet slot containing the private key for the host
- **ecdh_keypair_slot** id of the iot-safe applet slot to store the ECDH keypair
- **peer_pubkey_slot** id of the iot-safe applet slot to store the other endpoint's public key for ECDH
- **peer_cert_slot** id of the iot-safe applet slot to store the other endpoint's public key for verification

See:

- wolfSSL_iotsafe_on_ex

- wolfSSL_CTX_iotsafe_enable

Return:

- 0 upon success
- NOT_COMPILED_IN if HAVE_PK_CALLBACKS is disabled
- BAD_FUNC_ARG if the ssl pointer is invalid

This should be called to connect a SSL session to IoT-Safe applet when the ID of the slots are one-byte long. If IoT-SAFE slots have an ID of two or more bytes, wolfSSL_iotsafe_on_ex() should be used instead.

Example

```
// Define key ids for IoT-Safe
#define PRIVKEY_ID 0x02
#define ECDH_KEYPAIR_ID 0x03
#define PEER_PUBKEY_ID 0x04
#define PEER_CERT_ID 0x05
// Create new ssl session
WOLFSSL *ssl;
ssl = wolfSSL_new(ctx);
if (!ssl)
    return NULL;
// Enable IoT-Safe and associate key slots
ret = wolfSSL_CTX_iotsafe_enable(ctx);
if (ret == 0) {
    ret = wolfSSL_iotsafe_on(ssl, PRIVKEY_ID, ECDH_KEYPAIR_ID, PEER_PUBKEY_ID,
 ↪  PEER_CERT_ID);
}
```

C.30.2.3 function wolfSSL_iotsafe_on_ex

```
int wolfSSL_iotsafe_on_ex(
    WOLFSSL * ssl,
    byte * privkey_id,
    byte * ecdh_keypair_slot,
    byte * peer_pubkey_slot,
    byte * peer_cert_slot,
    word16 id_size
)
```

This function connects the IoT-Safe TLS callbacks to the given SSL session. This is equivalent to wolfSSL_iotsafe_on except that the IDs for the IoT-SAFE slots can be passed by reference, and the length of the ID fields can be specified via the parameter “id_size”.

Parameters:

- **ssl** pointer to the WOLFSSL object where the callbacks will be enabled
- **privkey_id** pointer to the id of the iot-safe applet slot containing the private key for the host
- **ecdh_keypair_slot** pointer to the id of the iot-safe applet slot to store the ECDH keypair
- **peer_pubkey_slot** pointer to the of id the iot-safe applet slot to store the other endpoint’s public key for ECDH
- **peer_cert_slot** pointer to the id of the iot-safe applet slot to store the other endpoint’s public key for verification
- **id_size** size of each slot ID

See:

- wolfSSL_iotsafe_on
- wolfSSL_CTX_iotsafe_enable

Return:

- 0 upon success
- NOT_COMPILED_IN if HAVE_PK_CALLBACKS is disabled
- BAD_FUNC_ARG if the ssl pointer is invalid

Example

```
// Define key ids for IoT-Safe (16 bit, little endian)
#define PRIVKEY_ID 0x0201
#define ECDH_KEYPAIR_ID 0x0301
#define PEER_PUBKEY_ID 0x0401
#define PEER_CERT_ID 0x0501
#define ID_SIZE (sizeof(word16))

word16 privkey = PRIVKEY_ID,
         ecdh_keypair = ECDH_KEYPAIR_ID,
         peer_pubkey = PEER_PUBKEY_ID,
         peer_cert = PEER_CERT_ID;



// Create new ssl session
WOLFSSL *ssl;
ssl = wolfSSL_new(ctx);
if (!ssl)
    return NULL;
// Enable IoT-Safe and associate key slots
ret = wolfSSL_CTX_iotsafe_enable(ctx);
if (ret == 0) {
    ret = wolfSSL_CTX_iotsafe_on_ex(ssl, &privkey, &ecdh_keypair, &peer_pubkey,
 ↪  &peer_cert, ID_SIZE);
}
```

C.30.2.4 function wolfIoTSafe_SetCSIM_read_cb

```
void wolfIoTSafe_SetCSIM_read_cb(
    wolfSSL_IOTSafe_CSIM_read_cb rf
)
```

Associates a read callback for the AT+CSIM commands. This input function is usually associated to a read event of a UART channel communicating with the modem. The read callback associated is global and changes for all the contexts that use IoT-safe support at the same time. This function must be called from a single thread, during initialization, before any IoT-Safe operation is performed.

Parameters:

- **rf** Read callback associated to a UART read event. The callback function takes two arguments (buf, len) and return the number of characters read, up to len. When a newline is encountered, the callback should return the number of characters received so far, including the newline character.

See: wolfIoTSafe_SetCSIM_write_cb

Example

```
// USART read function, defined elsewhere
int usart_read(char *buf, int len);

wolfIoTSafe_SetCSIM_read_cb(usart_read);
```

C.30.2.5 function wolfIoTSafe_SetCSIM_write_cb

```
void wolfIoTSafe_SetCSIM_write_cb(
    wolfSSL_IOTSafe_CSIM_write_cb wf
)
```

Associates a write callback for the AT+CSIM commands. This output function is usually associated to a write event on a UART channel communicating with the modem. The write callback associated is global and changes for all the contexts that use IoT-safe support at the same time. This function must be called from a single thread, during initialization, before any IoT-Safe operation is performed.

Parameters:

- **wf** Write callback associated to a UART write event. The callback function takes two arguments (buf, len) and return the number of characters written, up to len.

See: wolfIoTSafe_SetCSIM_read_cb

Example

```
// USART write function, defined elsewhere
int usart_write(const char *buf, int len);
wolfIoTSafe_SetCSIM_write_cb(usart_write);
```

C.30.2.6 function wolfIoTSafe_GetRandom

```
int wolfIoTSafe_GetRandom(
    unsigned char * out,
    word32 sz
)
```

Generate a random buffer of given size, using the IoT-Safe function GetRandom. This function is automatically used by the wolfCrypt RNG object.

Parameters:

- **out** the buffer where the random sequence of bytes is stored.
- **sz** the size of the random sequence to generate, in bytes

Return: 0 upon success

C.30.2.7 function wolfIoTSafe_GetCert

```
int wolfIoTSafe_GetCert(
    uint8_t id,
    unsigned char * output,
    unsigned long sz
)
```

Import a certificate stored in a file on IoT-Safe applet, and store it locally in memory. Works with one-byte file ID field.

Parameters:

- **id** The file id in the IoT-Safe applet where the certificate is stored
- **output** the buffer where the certificate will be imported

- **sz** the maximum size available in the buffer output

Return:

- the length of the certificate imported
- < 0 in case of failure

Example

```
#define CRT_CLIENT_FILE_ID 0x03
unsigned char cert_buffer[2048];
// Get the certificate into the buffer
cert_buffer_size = wolfIoTSafe_GetCert(CRT_CLIENT_FILE_ID, cert_buffer, 2048);
if (cert_buffer_size < 1) {
    printf("Bad cli cert\n");
    return -1;
}
printf("Loaded Client certificate from IoT-Safe, size = %lu\n",
 ↪  cert_buffer_size);

// Use the certificate buffer as identity for the TLS client context
if (wolfSSL_CTX_use_certificate_buffer(cli_ctx, cert_buffer,
            cert_buffer_size, SSL_FILETYPE_ASN1) != SSL_SUCCESS) {
    printf("Cannot load client cert\n");
    return -1;
}
printf("Client certificate successfully imported.\n");
```

C.30.2.8 function wolfIoTSafe_GetCert_ex

```
int wolfIoTSafe_GetCert_ex(
    uint8_t * id,
    uint16_t id_sz,
    unsigned char * output,
    unsigned long sz
)
```

Import a certificate stored in a file on IoT-Safe applet, and store it locally in memory. Equivalent to wolfIoTSafe_GetCert, except that it can be invoked with a file ID of two or more bytes.

Parameters:

- **id** Pointer to the file id in the IoT-Safe applet where the certificate is stored
- **id_sz** Size of the file id in bytes
- **output** the buffer where the certificate will be imported
- **sz** the maximum size available in the buffer output

Return:

- the length of the certificate imported
- < 0 in case of failure

Example

```
#define CRT_CLIENT_FILE_ID 0x0302
#define ID_SIZE (sizeof(word16))
unsigned char cert_buffer[2048];
word16 client_file_id = CRT_CLIENT_FILE_ID;
```

```
// Get the certificate into the buffer
cert_buffer_size = wolfIoTSafe_GetCert_ex(&client_file_id, ID_SIZE,
 ↪  cert_buffer, 2048);
if (cert_buffer_size < 1) {
    printf("Bad cli cert\n");
    return -1;
}
printf("Loaded Client certificate from IoT-Safe, size = %lu\n",
 ↪  cert_buffer_size);

// Use the certificate buffer as identity for the TLS client context
if (wolfSSL_CTX_use_certificate_buffer(cli_ctx, cert_buffer,
            cert_buffer_size, SSL_FILETYPE_ASN1) != SSL_SUCCESS) {
    printf("Cannot load client cert\n");
    return -1;
}
printf("Client certificate successfully imported.\n");
```

C.30.2.9 function wc_iotsafe_ecc_import_public

```
int wc_iotsafe_ecc_import_public(
    ecc_key * key,
    byte key_id
)
```

Import an ECC 256-bit public key, stored in the IoT-Safe applet, into an ecc_key object.

Parameters:

- **key** the ecc_key object that will contain the key imported from the IoT-Safe applet
- **id** The key id in the IoT-Safe applet where the public key is stored

See:

- wc_iotsafe_ecc_export_public
- wc_iotsafe_ecc_export_private

Return:

- 0 upon success
- < 0 in case of failure

C.30.2.10 function wc_iotsafe_ecc_export_public

```
int wc_iotsafe_ecc_export_public(
    ecc_key * key,
    byte key_id
)
```

Export an ECC 256-bit public key, from ecc_key object to a writable public-key slot into the IoT-Safe applet.

Parameters:

- **key** the ecc_key object containing the key to be exported
- **id** The key id in the IoT-Safe applet where the public key will be stored

See:

- wc_iotsafe_ecc_import_public_ex
- wc_iotsafe_ecc_export_private

Return:

- 0 upon success
- < 0 in case of failure

C.30.2.11 function wc_iotsafe_ecc_export_public_ex

```
int wc_iotsafe_ecc_export_public_ex(
    ecc_key * key,
    byte * key_id,
    word16 id_size
)
```

Export an ECC 256-bit public key, from ecc_key object to a writable public-key slot into the IoT-Safe applet. Equivalent to wc_iotsafe_ecc_export_public, except that it can be invoked with a key ID of two or more bytes.

Parameters:

- **key** the ecc_key object containing the key to be exported
- **key_id** pointer to the key id in the IoT-Safe applet where the public key will be stored
- **id_size** the key id size in bytes

See:

- wc_iotsafe_ecc_export_public
- wc_iotsafe_ecc_import_public_ex
- wc_iotsafe_ecc_export_private_ex

Return:

- 0 upon success
- < 0 in case of failure

Example

```
ecc_key key;
word16 keyId = 0x0302;

wc_ecc_init(&key);
wc_ecc_make_key(&rng, 32, &key);

int ret = wc_iotsafe_ecc_export_public_ex(&key, (byte*)&keyId,
                                          sizeof(keyId));
if (ret != 0) {
    // error exporting public key
}
```

C.30.2.12 function wc_iotsafe_ecc_import_public_ex

```
int wc_iotsafe_ecc_import_public_ex(
    ecc_key * key,
    byte * key_id,
    word16 id_size
)
```

Export an ECC 256-bit public key, from ecc_key object to a writable public-key slot into the IoT-Safe applet. Equivalent to wc_iotsafe_ecc_import_public, except that it can be invoked with a key ID of two or more bytes.

Parameters:

- **key** the ecc_key object containing the key to be exported
- **id** The pointer to the key id in the IoT-Safe applet where the public key will be stored
- **id_size** The key id size

See:

- wc_iotsafe_ecc_import_public
- wc_iotsafe_ecc_export_private

Return:

- 0 upon success
- < 0 in case of failure

C.30.2.13 function wc_iotsafe_ecc_export_private

```
int wc_iotsafe_ecc_export_private(
    ecc_key * key,
    byte key_id
)
```

Export an ECC 256-bit key, from ecc_key object to a writable private-key slot into the IoT-Safe applet.

Parameters:

- **key** the ecc_key object containing the key to be exported
- **id** The key id in the IoT-Safe applet where the private key will be stored

See:

- wc_iotsafe_ecc_export_private_ex
- wc_iotsafe_ecc_import_public
- wc_iotsafe_ecc_export_public

Return:

- 0 upon success
- < 0 in case of failure

C.30.2.14 function wc_iotsafe_ecc_export_private_ex

```
int wc_iotsafe_ecc_export_private_ex(
    ecc_key * key,
    byte * key_id,
    word16 id_size
)
```

Export an ECC 256-bit key, from ecc_key object to a writable private-key slot into the IoT-Safe applet. Equivalent to wc_iotsafe_ecc_export_private, except that it can be invoked with a key ID of two or more bytes.

Parameters:

- **key** the ecc_key object containing the key to be exported
- **id** The pointer to the key id in the IoT-Safe applet where the private key will be stored
- **id_size** The key id size

See:

- wc_iotsafe_ecc_export_private
- wc_iotsafe_ecc_import_public
- wc_iotsafe_ecc_export_public

Return:

- 0 upon success
- < 0 in case of failure

C.30.2.15 function wc_iotsafe_ecc_sign_hash

```
int wc_iotsafe_ecc_sign_hash(
    byte * in,
    word32 inlen,
    byte * out,
    word32 * outlen,
    byte key_id
)
```

Sign a pre-computed HASH, using a private key previously stored, or pre-provisioned, in the IoT-Safe applet.

Parameters:

- **in** pointer to the buffer containing the message hash to sign
- **inlen** length of the message hash to sign
- **out** buffer in which to store the generated signature
- **outlen** max length of the output buffer. Will store the bytes
- **id** key id in the IoT-Safe applet for the slot containing the private key to sign the payload written to out upon successfully generating a message signature

See:

- wc_iotsafe_ecc_sign_hash_ex
- wc_iotsafe_ecc_verify_hash
- wc_iotsafe_ecc_gen_k

Return:

- 0 upon success
- < 0 in case of failure

C.30.2.16 function wc_iotsafe_ecc_sign_hash_ex

```
int wc_iotsafe_ecc_sign_hash_ex(
    byte * in,
    word32 inlen,
    byte * out,
    word32 * outlen,
    byte * key_id,
    word16 id_size
)
```

Sign a pre-computed HASH, using a private key previously stored, or pre-provisioned, in the IoT-Safe applet. Equivalent to wc_iotsafe_ecc_sign_hash, except that it can be invoked with a key ID of two or more bytes.

Parameters:

- **in** pointer to the buffer containing the message hash to sign
- **inlen** length of the message hash to sign
- **out** buffer in which to store the generated signature
- **outlen** max length of the output buffer. Will store the bytes
- **id** pointer to a key id in the IoT-Safe applet for the slot containing the private key to sign the payload written to out upon successfully generating a message signature
- **id_size** The key id size

See:

- wc_iotsafe_ecc_sign_hash
- wc_iotsafe_ecc_verify_hash
- wc_iotsafe_ecc_gen_k

Return:

- 0 upon success
- < 0 in case of failure

C.30.2.17 function wc_iotsafe_ecc_verify_hash

```
int wc_iotsafe_ecc_verify_hash(
    byte * sig,
    word32 siglen,
    byte * hash,
    word32 hashlen,
    int * res,
    byte key_id
)
```

Verify an ECC signature against a pre-computed HASH, using a public key previously stored, or pre-provisioned, in the IoT-Safe applet. Result is written to res. 1 is valid, 0 is invalid. Note: Do not use the return value to test for valid. Only use res.

Parameters:

- **sig** buffer containing the signature to verify
- **hash** The hash (message digest) that was signed
- **hashlen** The length of the hash (octets)
- **res** Result of signature, 1==valid, 0==invalid
- **key_id** The id of the slot where the public ECC key is stored in the IoT-Safe applet

See:

- wc_iotsafe_ecc_verify_hash_ex
- wc_iotsafe_ecc_sign_hash
- wc_iotsafe_ecc_gen_k

Return:

- 0 upon success (even if the signature is not valid)
- < 0 in case of failure.

C.30.2.18 function wc_iotsafe_ecc_verify_hash_ex

```
int wc_iotsafe_ecc_verify_hash_ex(
    byte * sig,
    word32 siglen,
    byte * hash,
    word32 hashlen,
```

```
    int * res,
    byte * key_id,
    word16 id_size
)
```

Verify an ECC signature against a pre-computed HASH, using a public key previously stored, or pre-provisioned, in the IoT-Safe applet. Result is written to res. 1 is valid, 0 is invalid. Note: Do not use the return value to test for valid. Only use res. Equivalent to wc_iotsafe_ecc_verify_hash, except that it can be invoked with a key ID of two or more bytes.

Parameters:

- **sig** buffer containing the signature to verify
- **hash** The hash (message digest) that was signed
- **hashlen** The length of the hash (octets)
- **res** Result of signature, 1==valid, 0==invalid
- **key_id** The id of the slot where the public ECC key is stored in the IoT-Safe applet
- **id_size** The key id size

See:

- wc_iotsafe_ecc_verify_hash
- wc_iotsafe_ecc_sign_hash
- wc_iotsafe_ecc_gen_k

Return:

- 0 upon success (even if the signature is not valid)
- < 0 in case of failure.

C.30.2.19 function wc_iotsafe_ecc_gen_k

```
int wc_iotsafe_ecc_gen_k(
    byte key_id
)
```

Generate an ECC 256-bit keypair and store it in a (writable) slot into the IoT-Safe applet.

Parameters:

- **key_id** The id of the slot where the ECC key pair is stored in the IoT-Safe applet.

See:

- wc_iotsafe_ecc_gen_k_ex
- wc_iotsafe_ecc_sign_hash
- wc_iotsafe_ecc_verify_hash

Return:

- 0 upon success
- < 0 in case of failure.

C.30.2.20 function wc_iotsafe_ecc_gen_k_ex

```
int wc_iotsafe_ecc_gen_k_ex(
    byte * key_id,
    word16 id_size
)
```

Generate an ECC 256-bit keypair and store it in a (writable) slot into the IoT-Safe applet. Equivalent to wc_iotsafe_ecc_gen_k, except that it can be invoked with a key ID of two or more bytes.

Parameters:

- **key_id** The id of the slot where the ECC key pair is stored in the IoT-Safe applet.
- **id_size** The key id size

See:

- wc_iotsafe_ecc_gen_k
- wc_iotsafe_ecc_sign_hash_ex
- wc_iotsafe_ecc_verify_hash_ex

Return:

- 0 upon success
- < 0 in case of failure.

C.30.3 Source code

```
int wolfSSL_CTX_iotsafe_enable(WOLFSSL_CTX *ctx);


int wolfSSL_iotsafe_on(WOLFSSL *ssl, byte privkey_id,
       byte ecdh_keypair_slot, byte peer_pubkey_slot, byte peer_cert_slot);


int wolfSSL_iotsafe_on_ex(WOLFSSL *ssl, byte *privkey_id,
       byte *ecdh_keypair_slot, byte *peer_pubkey_slot, byte *peer_cert_slot,
 ↪  word16 id_size);


void wolfIoTSafe_SetCSIM_read_cb(wolfSSL_IOTSafe_CSIM_read_cb rf);

void wolfIoTSafe_SetCSIM_write_cb(wolfSSL_IOTSafe_CSIM_write_cb wf);



int wolfIoTSafe_GetRandom(unsigned char* out, word32 sz);


int wolfIoTSafe_GetCert(uint8_t id, unsigned char *output, unsigned long sz);


int wolfIoTSafe_GetCert_ex(uint8_t *id, uint16_t id_sz, unsigned char *output,
 ↪  unsigned long sz);

int wc_iotsafe_ecc_import_public(ecc_key *key, byte key_id);

int wc_iotsafe_ecc_export_public(ecc_key *key, byte key_id);

int wc_iotsafe_ecc_export_public_ex(ecc_key *key, byte *key_id,
                                    word16 id_size);
```

```
int wc_iotsafe_ecc_import_public_ex(ecc_key *key, byte *key_id, word16
 ↪  id_size);

int wc_iotsafe_ecc_export_private(ecc_key *key, byte key_id);

int wc_iotsafe_ecc_export_private_ex(ecc_key *key, byte *key_id, word16
 ↪  id_size);

int wc_iotsafe_ecc_sign_hash(byte *in, word32 inlen, byte *out, word32 *outlen,
 ↪  byte key_id);

int wc_iotsafe_ecc_sign_hash_ex(byte *in, word32 inlen, byte *out, word32
 ↪  *outlen, byte *key_id, word16 id_size);

int wc_iotsafe_ecc_verify_hash(byte *sig, word32 siglen, byte *hash, word32
 ↪  hashlen, int *res, byte key_id);

int wc_iotsafe_ecc_verify_hash_ex(byte *sig, word32 siglen, byte *hash, word32
 ↪  hashlen, int *res, byte *key_id, word16 id_size);

int wc_iotsafe_ecc_gen_k(byte key_id);

int wc_iotsafe_ecc_gen_k_ex(byte *key_id, word16 id_size);
```

C.31 kdf.h

C.31.1 Functions

	Name
int	**wc_SRTP_KDF**(const byte * key, word32 keySz, const byte * salt, word32 saltSz, int kdrIdx, const byte * idx, byte * key1, word32 key1Sz, byte * key2, word32 key2Sz, byte * key3, word32 key3Sz)This function derives keys using SRTP KDF algorithm.
int	**wc_SRTCP_KDF**(const byte * key, word32 keySz, const byte * salt, word32 saltSz, int kdrIdx, const byte * idx, byte * key1, word32 key1Sz, byte * key2, word32 key2Sz, byte * key3, word32 key3Sz)This function derives keys using SRTCP KDF algorithm.
int	**wc_SRTP_KDF_label**(const byte * key, word32 keySz, const byte * salt, word32 saltSz, int kdrIdx, const byte * idx, byte label, byte * outKey, word32 outKeySz)This function derives a key with label using SRTP KDF algorithm.
int	**wc_SRTCP_KDF_label**(const byte * key, word32 keySz, const byte * salt, word32 saltSz, int kdrIdx, const byte * idx, byte label, byte * outKey, word32 outKeySz)This function derives key with label using SRTCP KDF algorithm.

	Name
int	wc_SRTP_KDF_kdr_to_idx(word32 kdr)This function converts a kdr value to an index to use in SRTP/SRTCP KDF API.
int	wc_KDA_KDF_onestep(const byte * z, word32 zSz, const byte * fixedInfo, word32 fixedInfoSz, word32 derivedSecretSz, enum wc_HashType hashType, byte * output, word32 outputSz)Performs the single_step key derivation function (KDF) as specified in SP800-56C option 1.

C.31.2 Functions Documentation

C.31.2.1 function wc_SRTP_KDF

```
int wc_SRTP_KDF(
    const byte * key,
    word32 keySz,
    const byte * salt,
    word32 saltSz,
    int kdrIdx,
    const byte * idx,
    byte * key1,
    word32 key1Sz,
    byte * key2,
    word32 key2Sz,
    byte * key3,
    word32 key3Sz
)
```

This function derives keys using SRTP KDF algorithm.

Parameters:

- **key** Key to use with encryption.
- **keySz** Size of key in bytes.
- **salt** Random non-secret value.
- **saltSz** Size of random in bytes.
- **kdrIdx** Key derivation rate. kdr = 0 when -1, otherwise kdr = 2^kdrIdx.
- **idx** Index value to XOR in.
- **key1** First key. Label value of 0x00.
- **key1Sz** Size of first key in bytes.
- **key2** Second key. Label value of 0x01.
- **key2Sz** Size of second key in bytes.
- **key3** Third key. Label value of 0x02.
- **key3Sz** Size of third key in bytes.

See:

- wc_SRTCP_KDF
- wc_SRTP_KDF_label
- wc_SRTCP_KDF_label
- wc_SRTP_KDF_kdr_to_idx

Return:

- 0 Returned upon successful key derivation.
- BAD_FUNC_ARG Returned when key or salt is NULL
- BAD_FUNC_ARG Returned when key length is not 16, 24 or 32.
- BAD_FUNC_ARG Returned when saltSz is larger than 14.
- BAD_FUNC_ARG Returned when kdrIdx is less than -1 or larger than 24.
- MEMORY_E on dynamic memory allocation failure.

Example

```
unsigned char key[16] = { ... };
unsigned char salt[14] = { ... };
unsigned char idx[6] = { ... };
unsigned char keyE[16];
unsigned char keyA[20];
unsigned char keyS[14];
int kdrIdx = 0; // Use all of index
int ret;

ret = wc_SRTP_KDF(key, sizeof(key), salt, sizeof(salt), kdrIdx, idx,
    keyE, sizeof(keyE), keyA, sizeof(keyA), keyS, sizeof(keyS));
if (ret != 0) {
    WOLFSSL_MSG("wc_SRTP_KDF failed");
}
```

C.31.2.2 function wc_SRTCP_KDF

```
int wc_SRTCP_KDF(
    const byte * key,
    word32 keySz,
    const byte * salt,
    word32 saltSz,
    int kdrIdx,
    const byte * idx,
    byte * key1,
    word32 key1Sz,
    byte * key2,
    word32 key2Sz,
    byte * key3,
    word32 key3Sz
)
```

This function derives keys using SRTCP KDF algorithm.

Parameters:

- **key** Key to use with encryption.
- **keySz** Size of key in bytes.
- **salt** Random non-secret value.
- **saltSz** Size of random in bytes.
- **kdrIdx** Key derivation rate. kdr = 0 when -1, otherwise kdr = 2^kdrIdx.
- **idx** Index value to XOR in.
- **key1** First key. Label value of 0x03 (WC_SRTCP_LABEL_ENCRYPTION).
- **key1Sz** Size of first key in bytes.
- **key2** Second key. Label value of 0x04 (WC_SRTCP_LABEL_MSG_AUTH).
- **key2Sz** Size of second key in bytes.
- **key3** Third key. Label value of 0x05 (WC_SRTCP_LABEL_SALT).
- **key3Sz** Size of third key in bytes.

See:

- wc_SRTP_KDF
- wc_SRTP_KDF_label
- wc_SRTCP_KDF_label
- wc_SRTP_KDF_kdr_to_idx

Return:

- 0 Returned upon successful key derivation.
- BAD_FUNC_ARG Returned when key or salt is NULL
- BAD_FUNC_ARG Returned when key length is not 16, 24 or 32.
- BAD_FUNC_ARG Returned when saltSz is larger than 14.
- BAD_FUNC_ARG Returned when kdrIdx is less than -1 or larger than 24.
- MEMORY_E on dynamic memory allocation failure.

Example

```
unsigned char key[16] = { ... };
unsigned char salt[14] = { ... };
unsigned char idx[4] = { ... };
unsigned char keyE[16];
unsigned char keyA[20];
unsigned char keyS[14];
int kdrIdx = 0; // Use all of index
int ret;

ret = wc_SRTCP_KDF(key, sizeof(key), salt, sizeof(salt), kdrIdx, idx,
    keyE, sizeof(keyE), keyA, sizeof(keyA), keyS, sizeof(keyS));
if (ret != 0) {
    WOLFSSL_MSG("wc_SRTP_KDF failed");
}
```

C.31.2.3 function wc_SRTP_KDF_label

```
int wc_SRTP_KDF_label(
    const byte * key,
    word32 keySz,
    const byte * salt,
    word32 saltSz,
    int kdrIdx,
    const byte * idx,
    byte label,
    byte * outKey,
    word32 outKeySz
)
```

This function derives a key with label using SRTP KDF algorithm.

Parameters:

- **key** Key to use with encryption.
- **keySz** Size of key in bytes.
- **salt** Random non-secret value.
- **saltSz** Size of random in bytes.
- **kdrIdx** Key derivation rate. kdr = 0 when -1, otherwise kdr = 2^kdrIdx.
- **idx** Index value to XOR in.
- **label** Label to use when deriving key.

- **outKey** Derived key.
- **outKeySz** Size of derived key in bytes.

See:

- wc_SRTP_KDF
- wc_SRTCP_KDF
- wc_SRTCP_KDF_label
- wc_SRTP_KDF_kdr_to_idx

Return:

- 0 Returned upon successful key derivation.
- BAD_FUNC_ARG Returned when key, salt or outKey is NULL
- BAD_FUNC_ARG Returned when key length is not 16, 24 or 32.
- BAD_FUNC_ARG Returned when saltSz is larger than 14.
- BAD_FUNC_ARG Returned when kdrIdx is less than -1 or larger than 24.
- MEMORY_E on dynamic memory allocation failure.

Example

```
unsigned char key[16] = { ... };
unsigned char salt[14] = { ... };
unsigned char idx[6] = { ... };
unsigned char keyE[16];
int kdrIdx = 0; // Use all of index
int ret;

ret = wc_SRTP_KDF_label(key, sizeof(key), salt, sizeof(salt), kdrIdx, idx,
    WC_SRTP_LABEL_ENCRYPTION, keyE, sizeof(keyE));
if (ret != 0) {
    WOLFSSL_MSG("wc_SRTP_KDF failed");
}
```

C.31.2.4 function wc_SRTCP_KDF_label

```
int wc_SRTCP_KDF_label(
    const byte * key,
    word32 keySz,
    const byte * salt,
    word32 saltSz,
    int kdrIdx,
    const byte * idx,
    byte label,
    byte * outKey,
    word32 outKeySz
)
```

This function derives key with label using SRTCP KDF algorithm.

Parameters:

- **key** Key to use with encryption.
- **keySz** Size of key in bytes.
- **salt** Random non-secret value.
- **saltSz** Size of random in bytes.
- **kdrIdx** Key derivation rate. kdr = 0 when -1, otherwise kdr = 2^kdrIdx.
- **idx** Index value to XOR in.
- **label** Label to use when deriving key.

- **outKey** Derived key.
- **outKeySz** Size of derived key in bytes.

See:

- wc_SRTP_KDF
- wc_SRTCP_KDF
- wc_SRTP_KDF_label
- wc_SRTP_KDF_kdr_to_idx

Return:

- 0 Returned upon successful key derivation.
- BAD_FUNC_ARG Returned when key, salt or outKey is NULL
- BAD_FUNC_ARG Returned when key length is not 16, 24 or 32.
- BAD_FUNC_ARG Returned when saltSz is larger than 14.
- BAD_FUNC_ARG Returned when kdrIdx is less than -1 or larger than 24.
- MEMORY_E on dynamic memory allocation failure.

Example

```
unsigned char key[16] = { ... };
unsigned char salt[14] = { ... };
unsigned char idx[4] = { ... };
unsigned char keyE[16];
int kdrIdx = 0; // Use all of index
int ret;

ret = wc_SRTCP_KDF_label(key, sizeof(key), salt, sizeof(salt), kdrIdx,
    idx, WC_SRTCP_LABEL_ENCRYPTION, keyE, sizeof(keyE));
if (ret != 0) {
    WOLFSSL_MSG("wc_SRTP_KDF failed");
}
```

C.31.2.5 function wc_SRTP_KDF_kdr_to_idx

```
int wc_SRTP_KDF_kdr_to_idx(
    word32 kdr
)
```

This function converts a kdr value to an index to use in SRTP/SRTCP KDF API.

Parameters:

- **kdr** Key derivation rate to convert.

See:

- wc_SRTP_KDF
- wc_SRTCP_KDF
- wc_SRTP_KDF_label
- wc_SRTCP_KDF_label

Return: Key derivation rate as an index.

Example

```
word32 kdr = 0x00000010;
int kdrIdx;
int ret;
```

```
kdrIdx = wc_SRTP_KDF_kdr_to_idx(kdr);
```

C.31.2.6 function wc_KDA_KDF_onestep

```
int wc_KDA_KDF_onestep(
    const byte * z,
    word32 zSz,
    const byte * fixedInfo,
    word32 fixedInfoSz,
    word32 derivedSecretSz,
    enum wc_HashType hashType,
    byte * output,
    word32 outputSz
)
```

Performs the single-step key derivation function (KDF) as specified in SP800-56C option 1.

Parameters:

- **z** The input keying material.
- **zSz** The size of the input keying material.
- **fixedInfo** The fixed information to be included in the KDF.
- **fixedInfoSz** The size of the fixed information.
- **derivedSecretSz** The desired size of the derived secret.
- **hashType** The hash algorithm to be used in the KDF.
- **output** The buffer to store the derived secret.
- **outputSz** The size of the output buffer.

Return:

- 0 if the KDF operation is successful,
- BAD_FUNC_ARG if the input parameters are invalid.
- negative error code if the KDF operation fails.

Example

```
unsigned char z[32] = { ... };
unsigned char fixedInfo[16] = { ... };
unsigned char output[32];
int ret;

ret = wc_KDA_KDF_onestep(z, sizeof(z), fixedInfo, sizeof(fixedInfo),
    sizeof(output), WC_HASH_TYPE_SHA256, output, sizeof(output));
if (ret != 0) {
    WOLFSSL_MSG("wc_KDA_KDF_onestep failed");
}
```

C.31.3 Source code

```
int wc_SRTP_KDF(const byte* key, word32 keySz, const byte* salt, word32 saltSz,
        int kdrIdx, const byte* idx, byte* key1, word32 key1Sz, byte* key2,
        word32 key2Sz, byte* key3, word32 key3Sz);

int wc_SRTCP_KDF(const byte* key, word32 keySz, const byte* salt, word32
 ↪  saltSz,
        int kdrIdx, const byte* idx, byte* key1, word32 key1Sz, byte* key2,
```

```
        word32 key2Sz, byte* key3, word32 key3Sz);
int wc_SRTP_KDF_label(const byte* key, word32 keySz, const byte* salt,
        word32 saltSz, int kdrIdx, const byte* idx, byte label, byte* outKey,
        word32 outKeySz);
int wc_SRTCP_KDF_label(const byte* key, word32 keySz, const byte* salt,
        word32 saltSz, int kdrIdx, const byte* idx, byte label, byte* outKey,
        word32 outKeySz);
int wc_SRTP_KDF_kdr_to_idx(word32 kdr);

int wc_KDA_KDF_onestep(const byte* z, word32 zSz,
    const byte* fixedInfo, word32 fixedInfoSz, word32 derivedSecretSz,
    enum wc_HashType hashType, byte* output, word32 outputSz);
```

C.32 logging.h

C.32.1 Functions

	Name
int	**wolfSSL_SetLoggingCb**(wolfSSL_Logging_cb log_function)This function registers a logging callback that will be used to handle the wolfSSL log message. By default, if the system supports it fprintf() to stderr is used but by using this function anything can be done by the user.
int	**wolfSSL_Debugging_ON**(void)If logging has been enabled at build time this function turns on logging at runtime. To enable logging at build time use –enable-debug or define DEBUG_WOLFSSL.
void	**wolfSSL_Debugging_OFF**(void)This function turns off runtime logging messages. If they're already off, no action is taken.

C.32.2 Functions Documentation

C.32.2.1 function wolfSSL_SetLoggingCb

```
int wolfSSL_SetLoggingCb(
    wolfSSL_Logging_cb log_function
)
```

This function registers a logging callback that will be used to handle the wolfSSL log message. By default, if the system supports it fprintf() to stderr is used but by using this function anything can be done by the user.

Parameters:

- **log_function** function to register as a logging callback. Function signature must follow the above prototype.

See:

- wolfSSL_Debugging_ON
- wolfSSL_Debugging_OFF

Return:

- Success If successful this function will return 0.
- BAD_FUNC_ARG is the error that will be returned if a function pointer is not provided.

Example

```
int ret = 0;
// Logging callback prototype
void MyLoggingCallback(const int logLevel, const char* const logMessage);
// Register the custom logging callback with wolfSSL
ret = wolfSSL_SetLoggingCb(MyLoggingCallback);
if (ret != 0) {
    // failed to set logging callback
}
void MyLoggingCallback(const int logLevel, const char* const logMessage)
{
// custom logging function
}
```

C.32.2.2 function wolfSSL_Debugging_ON

```
int wolfSSL_Debugging_ON(
    void
)
```

If logging has been enabled at build time this function turns on logging at runtime. To enable logging at build time use –enable-debug or define DEBUG_WOLFSSL.

Parameters:

- **none** No parameters.

See:

- wolfSSL_Debugging_OFF
- wolfSSL_SetLoggingCb

Return:

- 0 upon success.
- NOT_COMPILED_IN is the error that will be returned if logging isn't enabled for this build.

Example

```
wolfSSL_Debugging_ON();
```

C.32.2.3 function wolfSSL_Debugging_OFF

```
void wolfSSL_Debugging_OFF(
    void
)
```

This function turns off runtime logging messages. If they're already off, no action is taken.

Parameters:

- **none** No parameters.

See:

- wolfSSL_Debugging_ON
- wolfSSL_SetLoggingCb

Return: none No returns.

Example

```
wolfSSL_Debugging_OFF();
```

C.32.3 Source code

```
int wolfSSL_SetLoggingCb(wolfSSL_Logging_cb log_function);

int  wolfSSL_Debugging_ON(void);

void wolfSSL_Debugging_OFF(void);
```

C.33 md2.h

C.33.1 Functions

	Name
int	**wc_InitMd2**(wc_Md2 * md2)This function initializes md2. This is automatically called by wc_Md2Hash.
int	**wc_Md2Update**(wc_Md2 * md2, const byte * data, word32 len)Can be called to continually hash the provided byte array of length len.
int	**wc_Md2Final**(wc_Md2 * md2, byte * hash)Finalizes hashing of data. Result is placed into hash.
int	**wc_Md2Hash**(const byte * data, word32 len, byte * hash)Convenience function, handles all the hashing and places the result into hash.

C.33.2 Functions Documentation

C.33.2.1 function wc_InitMd2

```
int wc_InitMd2(
    wc_Md2 * md2
)
```

This function initializes md2. This is automatically called by wc_Md2Hash.

Parameters:

- **md2** pointer to the md2 structure to use for encryption

See:

- wc_Md2Hash
- wc_Md2Update
- wc_Md2Final

Return:

- 0 Returned upon successfully initializing
- BAD_FUNC_ARG Returned if md2 is NULL

Example

```
md2 md2[1];
if ((ret = wc_InitMd2(md2)) != 0) {
   WOLFSSL_MSG("wc_Initmd2 failed");
}
else {
   wc_Md2Update(md2, data, len);
   wc_Md2Final(md2, hash);
}
```

C.33.2.2 function wc_Md2Update

```
int wc_Md2Update(
    wc_Md2 * md2,
    const byte * data,
    word32 len
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **md2** pointer to the md2 structure to use for encryption
- **data** the data to be hashed
- **len** length of data to be hashed

See:

- wc_Md2Hash
- wc_Md2Final
- wc_InitMd2

Return:

- 0 Returned upon successfully adding the data to the digest.
- BAD_FUNC_ARG Returned if md2 is NULL, or if data is NULL and len is non-zero

Example

```
md2 md2[1];
byte data[] = { }; // Data to be hashed
word32 len = sizeof(data);

if ((ret = wc_InitMd2(md2)) != 0) {
   WOLFSSL_MSG("wc_Initmd2 failed");
}
else {
   wc_Md2Update(md2, data, len);
   wc_Md2Final(md2, hash);
}
```

C.33.2.3 function wc_Md2Final

```
int wc_Md2Final(
    wc_Md2 * md2,
    byte * hash
)
```

Finalizes hashing of data. Result is placed into hash.

Parameters:

- **md2** pointer to the md2 structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Md2Hash
- wc_Md2Final
- wc_InitMd2

Return:

- 0 Returned upon successfully finalizing.
- BAD_FUNC_ARG Returned if md2 or hash is NULL

Example

```
md2 md2[1];
byte data[] = { }; // Data to be hashed
word32 len = sizeof(data);

if ((ret = wc_InitMd2(md2)) != 0) {
   WOLFSSL_MSG("wc_Initmd2 failed");
}
else {
   wc_Md2Update(md2, data, len);
   wc_Md2Final(md2, hash);
}
```

C.33.2.4 function wc_Md2Hash

```
int wc_Md2Hash(
    const byte * data,
    word32 len,
    byte * hash
)
```

Convenience function, handles all the hashing and places the result into hash.

Parameters:

- **data** the data to hash
- **len** the length of data
- **hash** Byte array to hold hash value.

See:

- wc_Md2Hash
- wc_Md2Final
- wc_InitMd2

Return:

- 0 Returned upon successfully hashing the data.
- Memory_E memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Example

none

C.33.3 Source code

```
int wc_InitMd2(wc_Md2* md2);

int wc_Md2Update(wc_Md2* md2, const byte* data, word32 len);

int wc_Md2Final(wc_Md2* md2, byte* hash);

int  wc_Md2Hash(const byte* data, word32 len, byte* hash);
```

C.34 md4.h

C.34.1 Functions

	Name
int	**wc_InitMd4**(wc_Md4 * md4)This function initializes md4. This is automatically called by wc_Md4Hash.
int	**wc_Md4Update**(wc_Md4 * md4, const byte * data, word32 len)Can be called to continually hash the provided byte array of length len.
int	**wc_Md4Final**(wc_Md4 * md4, byte * hash)Finalizes hashing of data. Result is placed into hash.

C.34.2 Functions Documentation

C.34.2.1 function wc_InitMd4

```
int wc_InitMd4(
    wc_Md4 * md4
)
```

This function initializes md4. This is automatically called by wc_Md4Hash.

Parameters:

- **md4** pointer to the md4 structure to use for encryption

See:

- wc_Md4Hash
- wc_Md4Update
- wc_Md4Final

Return:

- 0 Returned upon successfully initializing
- BAD_FUNC_ARG Returned if md4 is NULL

Example

```
md4 md4[1];
if ((ret = wc_InitMd4(md4)) != 0) {
   WOLFSSL_MSG("wc_Initmd4 failed");
}
```

```
else {
   wc_Md4Update(md4, data, len);
   wc_Md4Final(md4, hash);
}
```

C.34.2.2 function wc_Md4Update

```
int wc_Md4Update(
    wc_Md4 * md4,
    const byte * data,
    word32 len
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **md4** pointer to the md4 structure to use for encryption
- **data** the data to be hashed
- **len** length of data to be hashed

See:

- wc_Md4Hash
- wc_Md4Final
- wc_InitMd4

Return:

- 0 Returned upon successfully adding the data to the digest.
- BAD_FUNC_ARG Returned if md4 is NULL, or if data is NULL and len is non-zero

Example

```
md4 md4[1];
byte data[] = { }; // Data to be hashed
word32 len = sizeof(data);

if ((ret = wc_InitMd4(md4)) != 0) {
   WOLFSSL_MSG("wc_Initmd4 failed");
}
else {
   wc_Md4Update(md4, data, len);
   wc_Md4Final(md4, hash);
}
```

C.34.2.3 function wc_Md4Final

```
int wc_Md4Final(
    wc_Md4 * md4,
    byte * hash
)
```

Finalizes hashing of data. Result is placed into hash.

Parameters:

- **md4** pointer to the md4 structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Md4Hash
- wc_Md4Final
- wc_InitMd4

Return:

- 0 Returned upon successfully finalizing.
- BAD_FUNC_ARG Returned if md4 or hash is NULL

Example

```
md4 md4[1];
if ((ret = wc_InitMd4(md4)) != 0) {
    WOLFSSL_MSG("wc_Initmd4 failed");
}
else {
    wc_Md4Update(md4, data, len);
    wc_Md4Final(md4, hash);
}
```

C.34.3 Source code

```
int wc_InitMd4(wc_Md4* md4);

int wc_Md4Update(wc_Md4* md4, const byte* data, word32 len);

int wc_Md4Final(wc_Md4* md4, byte* hash);
```

C.35 md5.h

C.35.1 Functions

	Name
int	**wc_InitMd5**(wc_Md5 * md5)This function initializes md5. This is automatically called by wc_Md5Hash.
int	**wc_Md5Update**(wc_Md5 * md5, const byte * data, word32 len)Can be called to continually hash the provided byte array of length len.
int	**wc_Md5Final**(wc_Md5 * md5, byte * hash)Finalizes hashing of data. Result is placed into hash. Md5 Struct is reset. Note: This function will also return the result of calling IntelQaSymMd5() in the case that HAVE_INTEL_QA is defined.
void	**wc_Md5Free**(wc_Md5 * md5)Resets the Md5 structure. Note: this is only supported if you have WOLFSSL_TI_HASH defined.
int	**wc_Md5GetHash**(wc_Md5 * md5, byte * hash)Gets hash data. Result is placed into hash. Md5 struct is not reset.

C.35.2 Functions Documentation

C.35.2.1 function wc_InitMd5

```
int wc_InitMd5(
    wc_Md5 * md5
)
```

This function initializes md5. This is automatically called by wc_Md5Hash.

Parameters:

- **md5** pointer to the md5 structure to use for encryption

See:

- wc_Md5Hash
- wc_Md5Update
- wc_Md5Final

Return:

- 0 Returned upon successfully initializing.
- BAD_FUNC_ARG Returned if the Md5 structure is passed as a NULL value.

Example

```
Md5 md5;
byte* hash;
if ((ret = wc_InitMd5(&md5)) != 0) {
   WOLFSSL_MSG("wc_Initmd5 failed");
}
else {
   ret = wc_Md5Update(&md5, data, len);
   if (ret != 0) {
     // Md5 Update Failure Case.
   }
   ret = wc_Md5Final(&md5, hash);
  if (ret != 0) {
    // Md5 Final Failure Case.
  }
}
```

C.35.2.2 function wc_Md5Update

```
int wc_Md5Update(
    wc_Md5 * md5,
    const byte * data,
    word32 len
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **md5** pointer to the md5 structure to use for encryption
- **data** the data to be hashed
- **len** length of data to be hashed

See:

- wc_Md5Hash

- wc_Md5Final
- wc_InitMd5

Return:

- 0 Returned upon successfully adding the data to the digest.
- BAD_FUNC_ARG Returned if the Md5 structure is NULL or if data is NULL and len is greater than zero. The function should not return an error if the data parameter is NULL and len is zero.

Example

```
Md5 md5;
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitMd5(&md5)) != 0) {
   WOLFSSL_MSG("wc_Initmd5 failed");
}
else {
   ret = wc_Md5Update(&md5, data, len);
   if (ret != 0) {
     // Md5 Update Error Case.
   }
   ret = wc_Md5Final(&md5, hash);
   if (ret != 0) {
    // Md5 Final Error Case.
   }
}
```

C.35.2.3 function wc_Md5Final

```
int wc_Md5Final(
    wc_Md5 * md5,
    byte * hash
)
```

Finalizes hashing of data. Result is placed into hash. Md5 Struct is reset. Note: This function will also return the result of calling IntelQaSymMd5() in the case that HAVE_INTEL_QA is defined.

Parameters:

- **md5** pointer to the md5 structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Md5Hash
- wc_InitMd5
- wc_Md5GetHash

Return:

- 0 Returned upon successfully finalizing.
- BAD_FUNC_ARG Returned if the Md5 structure or hash pointer is passed in NULL.

Example

```
md5 md5[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);
```

```
if ((ret = wc_InitMd5(md5)) != 0) {
   WOLFSSL_MSG("wc_Initmd5 failed");
}
else {
   ret = wc_Md5Update(md5, data, len);
   if (ret != 0) {
    // Md5 Update Failure Case.
   }
  ret = wc_Md5Final(md5, hash);
   if (ret != 0) {
    // Md5 Final Failure Case.
   }
}
```

C.35.2.4 function wc_Md5Free

```
void wc_Md5Free(
    wc_Md5 * md5
)
```

Resets the Md5 structure. Note: this is only supported if you have WOLFSSL_TI_HASH defined.

Parameters:

- **md5** Pointer to the Md5 structure to be reset.

See:

- wc_InitMd5
- wc_Md5Update
- wc_Md5Final

Return: none No returns.

Example

```
Md5 md5;
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitMd5(&md5)) != 0) {
    WOLFSSL_MSG("wc_InitMd5 failed");
}
else {
    wc_Md5Update(&md5, data, len);
    wc_Md5Final(&md5, hash);
    wc_Md5Free(&md5);
}
```

C.35.2.5 function wc_Md5GetHash

```
int wc_Md5GetHash(
    wc_Md5 * md5,
    byte * hash
)
```

Gets hash data. Result is placed into hash. Md5 struct is not reset.

Parameters:

- **md5** pointer to the md5 structure to use for encryption.
- **hash** Byte array to hold hash value.

See:

- wc_Md5Hash
- wc_Md5Final
- wc_InitMd5

Return: none No returns

Example

```
md5 md5[1];
if ((ret = wc_InitMd5(md5)) != 0) {
   WOLFSSL_MSG("wc_Initmd5 failed");
}
else {
   wc_Md5Update(md5, data, len);
   wc_Md5GetHash(md5, hash);
}
```

C.35.3 Source code

```
int wc_InitMd5(wc_Md5* md5);

int wc_Md5Update(wc_Md5* md5, const byte* data, word32 len);

int wc_Md5Final(wc_Md5* md5, byte* hash);

void wc_Md5Free(wc_Md5* md5);

int  wc_Md5GetHash(wc_Md5* md5, byte* hash);
```

C.36 memory.h

C.36.1 Functions

	Name
void *	**wolfSSL_Malloc**(size_t size, void * heap, int type)This function is similar to malloc(), but calls the memory allocation function which wolfSSL has been configured to use. By default, wolfSSL uses malloc(). This can be changed using the wolfSSL memory abstraction layer _ see wolfSSL_SetAllocators(). Note wolfSSL_Malloc is not called directly by wolfSSL, but instead called by macro XMALLOC. For the default build only the size argument exists. If using WOLFSSL_STATIC_MEMORY build then heap and type arguments are included.

	Name
void	**wolfSSL_Free**(void * ptr, void * heap, int type)This function is similar to free(), but calls the memory free function which wolfSSL has been configured to use. By default, wolfSSL uses free(). This can be changed using the wolfSSL memory abstraction layer _ see wolfSSL_SetAllocators(). Note wolfSSL_Free is not called directly by wolfSSL, but instead called by macro XFREE. For the default build only the ptr argument exists. If using WOLFSSL_STATIC_MEMORY build then heap and type arguments are included.
void *	**wolfSSL_Realloc**(void * ptr, size_t size, void * heap, int type)This function is similar to realloc(), but calls the memory re_allocation function which wolfSSL has been configured to use. By default, wolfSSL uses realloc(). This can be changed using the wolfSSL memory abstraction layer _ see wolfSSL_SetAllocators(). Note wolfSSL_Realloc is not called directly by wolfSSL, but instead called by macro XREALLOC. For the default build only the size argument exists. If using WOLFSSL_STATIC_MEMORY build then heap and type arguments are included.
int	**wolfSSL_SetAllocators**(wolfSSL_Malloc_cb mf, wolfSSL_Free_cb ff, wolfSSL_Realloc_cb rf)This function registers the allocation functions used by wolfSSL. By default, if the system supports it, malloc/free and realloc are used. Using this function allows the user at runtime to install their own memory handlers.
int	**wolfSSL_StaticBufferSz**(byte * buffer, word32 sz, int flag)This function is available when static memory feature is used (–enable_staticmemory). It gives the optimum buffer size for memory “buckets”. This allows for a way to compute buffer size so that no extra unused memory is left at the end after it has been partitioned. For the none _ex version of this function the default bucket and distribution list set during compile time is used. The returned value, if positive, is the computed buffer size to use.
int	**wolfSSL_MemoryPaddingSz**(void)This function is available when static memory feature is used (–enable_staticmemory). It gives the size of padding needed for each partition of memory. This padding size will be the size needed to contain a memory management structure along with any extra for memory alignment.

	Name
int	**wolfSSL_CTX_load_static_memory**(WOLFSSL_CTX ** ctx, wolfSSL_method_func method, unsigned char * buf, unsigned int sz, int flag, int max)This function is used to set aside static memory for a CTX. Memory set aside is then used for the CTX's lifetime and for any SSL objects created from the CTX. By passing in a NULL ctx pointer and a wolfSSL_method_func function the creation of the CTX itself will also use static memory. wolfSSL_method_func has the function signature of WOLFSSL_METHOD* (*wolfSSL_method_func)(void* heap);. Passing in 0 for max makes it behave as if not set and no max concurrent use restrictions is in place. The flag value passed in determines how the memory is used and behavior while operating. Available flags are the following.
int	**wolfSSL_CTX_is_static_memory**(WOLFSSL_CTX * ctx, WOLFSSL_MEM_STATS * mem_stats)This function does not change any of the connections behavior and is used only for gathering information about the static memory usage.
int	**wolfSSL_is_static_memory**(WOLFSSL * ssl, WOLFSSL_MEM_CONN_STATS * mem_stats)wolfSSL_is_static_memory is used to gather information about a SSL's static memory usage. The return value indicates if static memory is being used and WOLFSSL_MEM_CONN_STATS will be filled out if and only if the flag WOLFMEM_TRACK_STATS was passed to the parent CTX when loading in static memory.
int	**wc_LoadStaticMemory**(WOLFSSL_HEAP_HINT ** pHint, unsigned char * buf, unsigned int sz, int flag, int max)This function is used to set aside static memory for wolfCrypt use. Memory can be used by passing the created heap hint into functions. An example of this is when calling wc_InitRng_ex. The flag value passed in determines how the memory is used and behavior while operating, in general wolfCrypt operations will use memory from a WOLFMEM_GENERAL pool. Available flags are the following.

	Name
int	**wc_LoadStaticMemory_ex**(WOLFSSL_HEAP_HINT ** pHint, unsigned int listSz, const word32 * sizeList, const word32 * distList, unsigned char * buf, unsigned int sz, int flag, int max)This function is used to set aside static memory for wolfCrypt use with custom bucket sizes and distributions. Memory can be used by passing the created heap hint into functions. This extended version allows for custom bucket sizes and distributions instead of using the default predefined sizes.
WOLFSSL_HEAP_HINT *	**wolfSSL_SetGlobalHeapHint**(WOLFSSL_HEAP_HINT * hint)This function sets a global heap hint that will be used when NULL heap hint is passed to memory allocation functions. This allows for setting a default heap hint that will be used across the entire application.
WOLFSSL_HEAP_HINT *	**wolfSSL_GetGlobalHeapHint**(void)This function gets the current global heap hint that is used when NULL heap hint is passed to memory allocation functions.
void	**wolfSSL_SetDebugMemoryCb**(DebugMemoryCb cb)This function sets a debug callback function for static memory allocation tracking. Used with WOLFSSL_STATIC_MEMORY_DEBUG_CALLBACK build option. The callback function will be called during memory allocation and deallocation operations to provide debugging information.
void	**wc_UnloadStaticMemory**(WOLFSSL_HEAP_HINT * heap)This function frees static memory heap and associated mutex. Should be called when done using static memory allocation to properly clean up resources.
int	**wolfSSL_StaticBufferSz_ex**(unsigned int listSz, const word32 * sizeList, const word32 * distList, byte * buffer, word32 sz, int flag)This function calculates the required buffer size for static memory allocation with custom bucket sizes and distributions. This extended version allows for custom bucket sizes instead of using the default predefined sizes.

C.36.2 Functions Documentation

C.36.2.1 function wolfSSL_Malloc

```
void * wolfSSL_Malloc(
    size_t size,
    void * heap,
    int type
```

)

This function is similar to malloc(), but calls the memory allocation function which wolfSSL has been configured to use. By default, wolfSSL uses malloc(). This can be changed using the wolfSSL memory abstraction layer - see wolfSSL_SetAllocators(). Note wolfSSL_Malloc is not called directly by wolfSSL, but instead called by macro XMALLOC. For the default build only the size argument exists. If using WOLFSSL_STATIC_MEMORY build then heap and type arguments are included.

Parameters:

- **size** size, in bytes, of the memory to allocate
- **heap** heap hint to use for memory. Can be NULL
- **type** dynamic type (see DYNAMIC_TYPE_ list in types.h)

See:

- wolfSSL_Free
- wolfSSL_Realloc
- wolfSSL_SetAllocators
- XMALLOC
- XFREE
- XREALLOC

Return:

- pointer If successful, this function returns a pointer to allocated memory.
- error If there is an error, NULL will be returned.

Example

```
int* tenInts = (int*)wolfSSL_Malloc(sizeof(int)*10);
```

C.36.2.2 function wolfSSL_Free

```
void wolfSSL_Free(
    void * ptr,
    void * heap,
    int type
)
```

This function is similar to free(), but calls the memory free function which wolfSSL has been configured to use. By default, wolfSSL uses free(). This can be changed using the wolfSSL memory abstraction layer - see wolfSSL_SetAllocators(). Note wolfSSL_Free is not called directly by wolfSSL, but instead called by macro XFREE. For the default build only the ptr argument exists. If using WOLFSSL_STATIC_MEMORY build then heap and type arguments are included.

Parameters:

- **ptr** pointer to the memory to be freed.
- **heap** heap hint to use for memory. Can be NULL
- **type** dynamic type (see DYNAMIC_TYPE_ list in types.h)

See:

- wolfSSL_Alloc
- wolfSSL_Realloc
- wolfSSL_SetAllocators
- XMALLOC
- XFREE
- XREALLOC

Return: none No returns.

Example

```
int* tenInts = (int*)wolfSSL_Malloc(sizeof(int)*10);
// process data as desired
...
if(tenInts) {
    wolfSSL_Free(tenInts, NULL, DYNAMIC_TYPE_TMP_BUFFER);
}
```

C.36.2.3 function wolfSSL_Realloc

```
void * wolfSSL_Realloc(
    void * ptr,
    size_t size,
    void * heap,
    int type
)
```

This function is similar to realloc(), but calls the memory re-allocation function which wolfSSL has been configured to use. By default, wolfSSL uses realloc(). This can be changed using the wolfSSL memory abstraction layer - see wolfSSL_SetAllocators(). Note wolfSSL_Realloc is not called directly by wolfSSL, but instead called by macro XREALLOC. For the default build only the size argument exists. If using WOLFSSL_STATIC_MEMORY build then heap and type arguments are included.

Parameters:

- **ptr** pointer to the previously-allocated memory, to be reallocated.
- **size** number of bytes to allocate.
- **heap** heap hint to use for memory. Can be NULL
- **type** dynamic type (see DYNAMIC_TYPE_ list in types.h)

See:

- wolfSSL_Free
- wolfSSL_Malloc
- wolfSSL_SetAllocators
- XMALLOC
- XFREE
- XREALLOC

Return:

- pointer If successful, this function returns a pointer to re-allocated memory. This may be the same pointer as ptr, or a new pointer location.
- Null If there is an error, NULL will be returned.

Example

```
int* tenInts = (int*)wolfSSL_Malloc(sizeof(int)*10);
int* twentyInts = (int*)wolfSSL_Realloc(tenInts, sizeof(int)*20);
```

C.36.2.4 function wolfSSL_SetAllocators

```
int wolfSSL_SetAllocators(
    wolfSSL_Malloc_cb mf,
    wolfSSL_Free_cb ff,
    wolfSSL_Realloc_cb rf
)
```

This function registers the allocation functions used by wolfSSL. By default, if the system supports it, malloc/free and realloc are used. Using this function allows the user at runtime to install their own memory handlers.

Parameters:

- **malloc_function** memory allocation function for wolfSSL to use. Function signature must match wolfSSL_Malloc_cb prototype, above.
- **free_function** memory free function for wolfSSL to use. Function signature must match wolfSSL_Free_cb prototype, above.
- **realloc_function** memory re-allocation function for wolfSSL to use. Function signature must match wolfSSL_Realloc_cb prototype, above.

See: none

Return:

- Success If successful this function will return 0.
- BAD_FUNC_ARG is the error that will be returned if a function pointer is not provided.

Example

```
static void* MyMalloc(size_t size)
{
    // custom malloc function
}

static void MyFree(void* ptr)
{
    // custom free function
}

static void* MyRealloc(void* ptr, size_t size)
{
    // custom realloc function
}

// Register custom memory functions with wolfSSL
int ret = wolfSSL_SetAllocators(MyMalloc, MyFree, MyRealloc);
if (ret != 0) {
    // failed to set memory functions
}
```

C.36.2.5 function wolfSSL_StaticBufferSz

```
int wolfSSL_StaticBufferSz(
    byte * buffer,
    word32 sz,
    int flag
)
```

This function is available when static memory feature is used (–enable-staticmemory). It gives the optimum buffer size for memory “buckets”. This allows for a way to compute buffer size so that no extra unused memory is left at the end after it has been partitioned. For the none _ex version of this function the default bucket and distribution list set during compile time is used. The returned value, if positive, is the computed buffer size to use.

Parameters:

- **buffer** pointer to buffer
- **size** size of buffer
- **type** desired type of memory ie WOLFMEM_GENERAL or WOLFMEM_IO_POOL

See:

- wolfSSL_Malloc
- wolfSSL_Free

Return:

- Success On successfully completing buffer size calculations a positive value is returned. This returned value is for optimum buffer size.
- Failure All negative values are considered to be error cases.

Example

```
byte buffer[1000];
word32 size = sizeof(buffer);
int optimum;

optimum = wolfSSL_StaticBufferSz(buffer, size, WOLFMEM_GENERAL);
if (optimum < 0) { //handle error case }
printf("The optimum buffer size to make use of all memory is %d\n",
optimum);
...
```

C.36.2.6 function wolfSSL_MemoryPaddingSz

```
int wolfSSL_MemoryPaddingSz(
    void
)
```

This function is available when static memory feature is used (–enable-staticmemory). It gives the size of padding needed for each partition of memory. This padding size will be the size needed to contain a memory management structure along with any extra for memory alignment.

Parameters:

- **none** No parameters.

See:

- wolfSSL_Malloc
- wolfSSL_Free

Return:

- On successfully memory padding calculation the return value will be a positive value
- All negative values are considered error cases.

Example

```
int padding;
padding = wolfSSL_MemoryPaddingSz();
if (padding < 0) { //handle error case }
printf("The padding size needed for each \"bucket\" of memory is %d\n",
padding);
// calculation of buffer for IO POOL size is number of buckets
// times (padding + WOLFMEM_IO_SZ)
...
```

C.36.2.7 function wolfSSL_CTX_load_static_memory

```
int wolfSSL_CTX_load_static_memory(
    WOLFSSL_CTX ** ctx,
    wolfSSL_method_func method,
    unsigned char * buf,
    unsigned int sz,
    int flag,
    int max
)
```

This function is used to set aside static memory for a CTX. Memory set aside is then used for the CTX's lifetime and for any SSL objects created from the CTX. By passing in a NULL ctx pointer and a wolfSSL_method_func function the creation of the CTX itself will also use static memory. wolfSSL_method_func has the function signature of WOLFSSL_METHOD* (*wolfSSL_method_func)(void* heap);. Passing in 0 for max makes it behave as if not set and no max concurrent use restrictions is in place. The flag value passed in determines how the memory is used and behavior while operating. Available flags are the following.

Parameters:

- **ctx** address of pointer to a WOLFSSL_CTX structure.
- **method** function to create protocol. (should be NULL if ctx is not also NULL)
- **buf** memory to use for all operations.
- **sz** size of memory buffer being passed in.
- **flag** type of memory.
- **max** max concurrent operations.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_is_static_memory
- wolfSSL_is_static_memory

Return:

- If successful, SSL_SUCCESS will be returned.
- All unsuccessful return values will be less than 0 or equal to SSL_FAILURE.

0 - default general memory

WOLFMEM_IO_POOL - used for input/output buffer when sending receiving messages. Overrides general memory, so all memory in buffer passed in is used for IO. WOLFMEM_IO_FIXED - same as WOLFMEM_IO_POOL but each SSL now keeps two buffers to themselves for their lifetime. WOLFMEM_TRACK_STATS - each SSL keeps track of memory stats while running.

Example

```
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
int ret;
unsigned char memory[MAX];
int memorySz = MAX;
unsigned char IO[MAX];
int IOSz = MAX;
int flag = WOLFMEM_IO_FIXED | WOLFMEM_TRACK_STATS;
...
// create ctx also using static memory, start with general memory to use
ctx = NULL:
```

```
ret = wolfSSL_CTX_load_static_memory(&ctx, wolfSSLv23_server_method_ex,
 ↪  memory, memorySz, 0,
MAX_CONCURRENT_HANDSHAKES);
if (ret != SSL_SUCCESS) {
// handle error case
}
// load in memory for use with IO
ret = wolfSSL_CTX_load_static_memory(&ctx, NULL, IO, IOSz, flag,
 ↪  MAX_CONCURRENT_IO);
if (ret != SSL_SUCCESS) {
// handle error case
}
...
```

C.36.2.8 function wolfSSL_CTX_is_static_memory

```
int wolfSSL_CTX_is_static_memory(
    WOLFSSL_CTX * ctx,
    WOLFSSL_MEM_STATS * mem_stats
)
```

This function does not change any of the connections behavior and is used only for gathering information about the static memory usage.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **mem_stats** structure to hold information about staic memory usage.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_load_static_memory
- wolfSSL_is_static_memory

Return:

- A value of 1 is returned if using static memory for the CTX is true.
- 0 is returned if not using static memory.

Example

```
WOLFSSL_CTX* ctx;
int ret;
WOLFSSL_MEM_STATS mem_stats;
...
//get information about static memory with CTX

ret = wolfSSL_CTX_is_static_memory(ctx, &mem_stats);

if (ret == 1) {
    // handle case of is using static memory
    // print out or inspect elements of mem_stats
}

if (ret == 0) {
    //handle case of ctx not using static memory
}
...
```

C.36.2.9 function wolfSSL_is_static_memory

```
int wolfSSL_is_static_memory(
    WOLFSSL * ssl,
    WOLFSSL_MEM_CONN_STATS * mem_stats
)
```

wolfSSL_is_static_memory is used to gather information about a SSL's static memory usage. The return value indicates if static memory is being used and WOLFSSL_MEM_CONN_STATS will be filled out if and only if the flag WOLFMEM_TRACK_STATS was passed to the parent CTX when loading in static memory.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **mem_stats** structure to contain static memory usage

See:

- wolfSSL_new
- wolfSSL_CTX_is_static_memory

Return:

- A value of 1 is returned if using static memory for the CTX is true.
- 0 is returned if not using static memory.

Example

```
WOLFSSL* ssl;
int ret;
WOLFSSL_MEM_CONN_STATS mem_stats;

...

ret = wolfSSL_is_static_memory(ssl, mem_stats);

if (ret == 1) {
    // handle case when is static memory
    // investigate elements in mem_stats if WOLFMEM_TRACK_STATS flag
}
...
```

C.36.2.10 function wc_LoadStaticMemory

```
int wc_LoadStaticMemory(
    WOLFSSL_HEAP_HINT ** pHint,
    unsigned char * buf,
    unsigned int sz,
    int flag,
    int max
)
```

This function is used to set aside static memory for wolfCrypt use. Memory can be used by passing the created heap hint into functions. An example of this is when calling wc_InitRng_ex. The flag value passed in determines how the memory is used and behavior while operating, in general wolfCrypt operations will use memory from a WOLFMEM_GENERAL pool. Available flags are the following.

Parameters:

- **pHint** WOLFSSL_HEAP_HINT structure to use
- **buf** memory to use for all operations.

- **sz** size of memory buffer being passed in.
- **flag** type of memory.
- **max** max concurrent operations (handshakes, IO).

See: none

Return:

- Returns 0 on success.
- Returns a non-zero integer on failure.

WOLFMEM_GENERAL - default general memory

WOLFMEM_IO_POOL - used for input/output buffer when sending receiving messages. Overrides general memory, so all memory in buffer passed in is used for IO. WOLFMEM_IO_FIXED - same as WOLFMEM_IO_POOL but each SSL now keeps two buffers to themselves for their lifetime. WOLFMEM_TRACK_STATS - each SSL keeps track of memory stats while running

Example

```
WOLFSSL_HEAP_HINT hint;
int ret;
unsigned char memory[MAX];
int memorySz = MAX;
int flag = WOLFMEM_GENERAL | WOLFMEM_TRACK_STATS;
...

// load in memory for use

ret = wc_LoadStaticMemory(&hint, memory, memorySz, flag, 0);
if (ret) {
    // handle error case
}
...

ret = wc_InitRng_ex(&rng, hint, 0);

// check ret value
```

C.36.2.11 function wc_LoadStaticMemory_ex

```
int wc_LoadStaticMemory_ex(
    WOLFSSL_HEAP_HINT ** pHint,
    unsigned int listSz,
    const word32 * sizeList,
    const word32 * distList,
    unsigned char * buf,
    unsigned int sz,
    int flag,
    int max
)
```

This function is used to set aside static memory for wolfCrypt use with custom bucket sizes and distributions. Memory can be used by passing the created heap hint into functions. This extended version allows for custom bucket sizes and distributions instead of using the default predefined sizes.

Parameters:

- **pHint** WOLFSSL_HEAP_HINT handle to initialize

- **listSz** number of entries in the size and distribution lists
- **sizeList** array of bucket sizes to use
- **distList** distribution list matching sizeList
- **buf** memory to use for all operations.
- **sz** size of memory buffer being passed in.
- **flag** type of memory.
- **max** max concurrent operations (handshakes, IO).

See:

- wc_LoadStaticMemory
- wc_UnloadStaticMemory

Return:

- Returns 0 on success.
- Returns a non-zero integer on failure.

Example

```
WOLFSSL_HEAP_HINT hint;
int ret;
unsigned char memory[MAX];
int memorySz = MAX;
int flag = WOLFMEM_GENERAL | WOLFMEM_TRACK_STATS;
const word32 sizeList[] = {64, 128, 256, 512, 1024};
const word32 distList[] = {1, 1, 1, 1, 1};
unsigned int listSz = (unsigned int)(sizeof(sizeList)/
                                     sizeof(sizeList[0]));
...

// load in memory for use with custom bucket sizes

ret = wc_LoadStaticMemory_ex(&hint, listSz, sizeList, distList,
                             memory, memorySz, flag, 0);
if (ret) {
    // handle error case
}
...

ret = wc_InitRng_ex(&rng, hint, 0);

// check ret value
```

C.36.2.12 function wolfSSL_SetGlobalHeapHint

```
WOLFSSL_HEAP_HINT * wolfSSL_SetGlobalHeapHint(
    WOLFSSL_HEAP_HINT * hint
)
```

This function sets a global heap hint that will be used when NULL heap hint is passed to memory allocation functions. This allows for setting a default heap hint that will be used across the entire application.

Parameters:

- **hint** WOLFSSL_HEAP_HINT structure to use as the global heap hint

See:

- wolfSSL_GetGlobalHeapHint
- wc_LoadStaticMemory

Return: Returns the previous global heap hint that was set.

Example

```
WOLFSSL_HEAP_HINT hint;
WOLFSSL_HEAP_HINT* prev_hint;
int ret;
unsigned char memory[MAX];
int memorySz = MAX;
...

// load in memory for use
ret = wc_LoadStaticMemory(&hint, memory, memorySz, WOLFMEM_GENERAL, 0);
if (ret != SSL_SUCCESS) {
    // handle error case
}

// set as global heap hint
prev_hint = wolfSSL_SetGlobalHeapHint(&hint);
if (prev_hint != NULL) {
    // there was a previous global heap hint
}
```

C.36.2.13 function wolfSSL_GetGlobalHeapHint

```
WOLFSSL_HEAP_HINT * wolfSSL_GetGlobalHeapHint(
    void
)
```

This function gets the current global heap hint that is used when NULL heap hint is passed to memory allocation functions.

Parameters:

- **none** No parameters.

See:

- wolfSSL_SetGlobalHeapHint
- wc_LoadStaticMemory

Return: Returns the current global heap hint, or NULL if none is set.

Example

```
WOLFSSL_HEAP_HINT* current_hint;
...

current_hint = wolfSSL_GetGlobalHeapHint();
if (current_hint != NULL) {
    // there is a global heap hint set
    // can use current_hint for operations
}
```

C.36.2.14 function wolfSSL_SetDebugMemoryCb

```
void wolfSSL_SetDebugMemoryCb(
    DebugMemoryCb cb
)
```

This function sets a debug callback function for static memory allocation tracking. Used with WOLFSSL_STATIC_MEMORY_DEBUG_CALLBACK build option. The callback function will be called during memory allocation and deallocation operations to provide debugging information.

Parameters:

- **cb** debug callback function to set

See: none

Return:

- If successful, 0 will be returned.
- All unsuccessful return values will be less than 0.

Example

```
static void debug_memory_cb(const char* func, const char* file, int line,
                            void* ptr, size_t size, int type)
{
    printf("Memory %s: %s:%d ptr=%p size=%zu type=%d\n",
           func, file, line, ptr, size, type);
}
...

// set debug callback
int ret = wolfSSL_SetDebugMemoryCb(debug_memory_cb);
if (ret != 0) {
    // handle error case
}
```

C.36.2.15 function wc_UnloadStaticMemory

```
void wc_UnloadStaticMemory(
    WOLFSSL_HEAP_HINT * heap
)
```

This function frees static memory heap and associated mutex. Should be called when done using static memory allocation to properly clean up resources.

Parameters:

- **hint** WOLFSSL_HEAP_HINT structure to unload

See:

- wc_LoadStaticMemory
- wc_LoadStaticMemory_ex

Return:

- If successful, 0 will be returned.
- All unsuccessful return values will be less than 0.

Example

```
WOLFSSL_HEAP_HINT hint;
int ret;
unsigned char memory[MAX];
```

```
int memorySz = MAX;
...

// load in memory for use
ret = wc_LoadStaticMemory(&hint, memory, memorySz, WOLFMEM_GENERAL, 0);
if (ret != SSL_SUCCESS) {
    // handle error case
}

// use memory for operations
...

// cleanup when done
wc_UnloadStaticMemory(&hint);
```

C.36.2.16 function wolfSSL_StaticBufferSz_ex

```
int wolfSSL_StaticBufferSz_ex(
    unsigned int listSz,
    const word32 * sizeList,
    const word32 * distList,
    byte * buffer,
    word32 sz,
    int flag
)
```

This function calculates the required buffer size for static memory allocation with custom bucket sizes and distributions. This extended version allows for custom bucket sizes instead of using the default predefined sizes.

Parameters:

- **bucket_sizes** array of bucket sizes to use
- **bucket_count** number of bucket sizes in the array
- **flag** desired type of memory ie WOLFMEM_GENERAL or WOLFMEM_IO_POOL

See:

- wolfSSL_StaticBufferSz
- wc_LoadStaticMemory_ex

Return:

- On successfully completing buffer size calculations a positive value is returned.
- All negative values are considered to be error cases.

Example

```
word32 sizeList[] = {64, 128, 256, 512, 1024};
word32 distList[] = {1, 2, 1, 1, 1};
int listSz = 5;
int optimum;

optimum = wolfSSL_StaticBufferSz_ex(listSz, sizeList, distList, NULL, 0,
    WOLFMEM_GENERAL);
if (optimum < 0) { //handle error case }
printf("The optimum buffer size with custom buckets is %d\n", optimum);
...
```

C.36.3 Source code

```
void* wolfSSL_Malloc(size_t size, void* heap, int type);

void  wolfSSL_Free(void *ptr, void* heap, int type);

void* wolfSSL_Realloc(void *ptr, size_t size, void* heap, int type);

int wolfSSL_SetAllocators(wolfSSL_Malloc_cb mf, wolfSSL_Free_cb ff,
                                    wolfSSL_Realloc_cb rf);

int wolfSSL_StaticBufferSz(byte* buffer, word32 sz, int flag);

int wolfSSL_MemoryPaddingSz(void);

int wolfSSL_CTX_load_static_memory(WOLFSSL_CTX** ctx, wolfSSL_method_func
 ↪  method,
        unsigned char* buf, unsigned int sz, int flag, int max);

int wolfSSL_CTX_is_static_memory(WOLFSSL_CTX* ctx, WOLFSSL_MEM_STATS*
 ↪  mem_stats);

int wolfSSL_is_static_memory(WOLFSSL* ssl, WOLFSSL_MEM_CONN_STATS* mem_stats);

int wc_LoadStaticMemory(WOLFSSL_HEAP_HINT** pHint, unsigned char* buf,
                unsigned int sz, int flag, int max);

int wc_LoadStaticMemory_ex(WOLFSSL_HEAP_HINT** pHint, unsigned int listSz,
                const word32 *sizeList, const word32 *distList,
                unsigned char* buf, unsigned int sz, int flag, int max);

WOLFSSL_HEAP_HINT* wolfSSL_SetGlobalHeapHint(WOLFSSL_HEAP_HINT* hint);

WOLFSSL_HEAP_HINT* wolfSSL_GetGlobalHeapHint(void);

void wolfSSL_SetDebugMemoryCb(DebugMemoryCb cb);

void wc_UnloadStaticMemory(WOLFSSL_HEAP_HINT* heap);

int wolfSSL_StaticBufferSz_ex(unsigned int listSz,
            const word32 *sizeList, const word32 *distList,
            byte* buffer, word32 sz, int flag);
```

C.37 ocsp.h

C.37.1 Functions

	Name
WOLFSSL_OCSP *	**wc_NewOCSP**(WOLFSSL_CERT_MANAGER * cm)Allocates and initialises an OCSP context.
void	**wc_FreeOCSP**(WOLFSSL_OCSP * ocsp)Frees resources associated with an OCSP context.

	Name
int	**wc_CheckCertOcspResponse**(WOLFSSL_OCSP * ocsp, DecodedCert * cert, byte * response, int responseSz, void * heap)Checks the OCSP response for a given certificate.
OcspRequest *	**wc_OcspRequest_new**(void * heap)Allocates a new OcspRequest structure.
void	**wc_OcspRequest_free**(OcspRequest * request)Frees an OcspRequest structure.
int	**wc_InitOcspRequest**(OcspRequest * req, DecodedCert * cert, byte useNonce, void * heap)Initializes an OcspRequest from a decoded certificate.
int	**wc_EncodeOcspRequest**(OcspRequest * req, byte * output, word32 size)Encodes an OCSP request into DER format.
OcspResponse *	**wc_OcspResponse_new**(void * heap)Allocates a new OcspResponse structure.
void	**wc_OcspResponse_free**(OcspResponse * response)Frees an OcspResponse structure.
OcspResponder *	**wc_OcspResponder_new**(void * heap, int sendCerts)Allocates and initializes a new OCSP Responder.
void	**wc_OcspResponder_free**(OcspResponder * responder)Frees an OCSP Responder and all associated resources.
int	**wc_OcspResponder_AddSigner**(OcspResponder * responder, const byte * signerDer, word32 signerDerSz, const byte * keyDer, word32 keyDerSz, const byte * issuerCertDer, word32 issuerCertDerSz)Adds a signing certificate and key to the OCSP Responder.
int	**wc_OcspResponder_SetCertStatus**(OcspResponder * responder, const char * caSubject, word32 caSubjectSz, const byte * serial, word32 serialSz, enum Ocsp_Cert_Status status, time_t revocationTime, enum WC_CRL_Reason revocationReason, word32 validityPeriod)Sets the revocation status of a certificate in the OCSP Responder.
int	**wc_OcspResponder_WriteResponse**(OcspResponder * responder, const byte * request, word32 requestSz, byte * response, word32 * responseSz)Generates an OCSP response for a given OCSP request.
int	**wc_OcspResponder_WriteErrorResponse**(enum Ocsp_Response_Status status, byte * response, word32 * responseSz)Generates an OCSP error response.

C.37.2 Functions Documentation

C.37.2.1 function wc_NewOCSP

```
WOLFSSL_OCSP * wc_NewOCSP(
    WOLFSSL_CERT_MANAGER * cm
)
```

Allocates and initialises an OCSP context.

Parameters:

- **cm** Pointer to the certificate manager.

See: wc_FreeOCSP

Return:

- Pointer to allocated WOLFSSL_OCSP on success
- NULL on failure

This function allocates and initialises a WOLFSSL_OCSP structure for use with OCSP operations.

C.37.2.2 function wc_FreeOCSP

```
void wc_FreeOCSP(
    WOLFSSL_OCSP * ocsp
)
```

Frees resources associated with an OCSP context.

Parameters:

- **ocsp** Pointer to the WOLFSSL_OCSP structure to free.

See: wc_NewOCSP

Return: void

This function releases any resources associated with a WOLFSSL_OCSP structure.

C.37.2.3 function wc_CheckCertOcspResponse

```
int wc_CheckCertOcspResponse(
    WOLFSSL_OCSP * ocsp,
    DecodedCert * cert,
    byte * response,
    int responseSz,
    void * heap
)
```

Checks the OCSP response for a given certificate.

Parameters:

- **ocsp** Pointer to the WOLFSSL_OCSP structure.
- **cert** Pointer to the decoded certificate.
- **response** Pointer to the OCSP response buffer.
- **responseSz** Size of the OCSP response buffer.
- **heap** Optional heap pointer.

Return:

- 0 on success
- <0 on failure

This function verifies an OCSP response for a specific certificate.

C.37.2.4 function wc_OcspRequest_new

```
OcspRequest * wc_OcspRequest_new(
    void * heap
)
```

Allocates a new OcspRequest structure.

Parameters:

- **heap** Pointer to a heap hint for dynamic memory allocation, or NULL to use the default allocator.

See:

- wc_OcspRequest_free
- wc_InitOcspRequest

Return:

- Pointer to a newly allocated OcspRequest on success.
- NULL on memory allocation failure.

This function allocates and zero-initializes an OcspRequest structure on the heap. The caller must free the returned object with wc_OcspRequest_free().

C.37.2.5 function wc_OcspRequest_free

```
void wc_OcspRequest_free(
    OcspRequest * request
)
```

Frees an OcspRequest structure.

Parameters:

- **request** Pointer to the OcspRequest to free. May be NULL, in which case this function is a no-op.

See: wc_OcspRequest_new

Return: void

This function releases all resources associated with an OcspRequest that was allocated with wc_OcspRequest_new(). It frees any internal allocations associated with the request before freeing the structure itself.

C.37.2.6 function wc_InitOcspRequest

```
int wc_InitOcspRequest(
    OcspRequest * req,
    DecodedCert * cert,
    byte useNonce,
    void * heap
)
```

Initializes an OcspRequest from a decoded certificate.

Parameters:

- **req** Pointer to the OcspRequest structure to initialize.
- **cert** Pointer to the DecodedCert from which to extract certificate information.
- **useNonce** If non-zero, a nonce extension will be added to the OCSP request.
- **heap** Pointer to a heap hint for dynamic memory allocation, or NULL.

See:

- wc_EncodeOcspRequest
- wc_OcspRequest_new
- wc_OcspRequest_free

Return:

- 0 on success.
- BAD_FUNC_ARG if req or cert is NULL.
- Other negative values on internal errors.

This function populates an OcspRequest structure with the issuer hash, issuer key hash, and serial number extracted from the given decoded certificate. Optionally, a nonce can be included in the request.

C.37.2.7 function wc_EncodeOcspRequest

```
int wc_EncodeOcspRequest(
    OcspRequest * req,
    byte * output,
    word32 size
)
```

Encodes an OCSP request into DER format.

Parameters:

- **req** Pointer to the initialized OcspRequest to encode.
- **output** Pointer to the output buffer that will receive the DER-encoded OCSP request.
- **size** Size of the output buffer in bytes.

See:

- wc_InitOcspRequest
- wc_OcspRequest_new

Return:

- The size of the encoded request in bytes on success.
- Negative error code on failure.

This function encodes a previously initialized OcspRequest into its DER wire format suitable for transmission to an OCSP responder.

C.37.2.8 function wc_OcspResponse_new

```
OcspResponse * wc_OcspResponse_new(
    void * heap
)
```

Allocates a new OcspResponse structure.

Parameters:

- **heap** Pointer to a heap hint for dynamic memory allocation, or NULL to use the default allocator.

See: wc_OcspResponse_free

Return:

- Pointer to a newly allocated OcspResponse on success.
- NULL on memory allocation failure.

This function allocates and zero-initializes an OcspResponse structure on the heap. The caller must free the returned object with wc_OcspResponse_free().

C.37.2.9 function wc_OcspResponse_free

```
void wc_OcspResponse_free(
    OcspResponse * response
)
```

Frees an OcspResponse structure.

Parameters:

- **response** Pointer to the OcspResponse to free. May be NULL, in which case this function is a no-op.

See: wc_OcspResponse_new

Return: void

This function releases all resources associated with an OcspResponse that was allocated with wc_OcspResponse_new(). It frees any internal allocations associated with the response before freeing the structure itself.

C.37.2.10 function wc_OcspResponder_new

```
OcspResponder * wc_OcspResponder_new(
    void * heap,
    int sendCerts
)
```

Allocates and initializes a new OCSP Responder.

Parameters:

- **heap** Pointer to a heap hint for dynamic memory allocation, or NULL.
- **sendCerts** If non-zero, the responder will include the signer certificate in generated OCSP responses.

See:

- wc_OcspResponder_free
- wc_OcspResponder_AddSigner
- wc_OcspResponder_SetCertStatus
- wc_OcspResponder_WriteResponse

Return:

- Pointer to a newly allocated OcspResponder on success.
- NULL on failure (memory allocation or RNG initialization failure).

This function creates an OcspResponder that can process OCSP requests and generate signed responses. The responder must have at least one signer added via wc_OcspResponder_AddSigner() before it can produce responses.

Requires HAVE_OCSP_RESPONDER to be defined.

C.37.2.11 function wc_OcspResponder_free

```
void wc_OcspResponder_free(
    OcspResponder * responder
)
```

Frees an OCSP Responder and all associated resources.

Parameters:

- **responder** Pointer to the OcspResponder to free. May be NULL, in which case this function is a no-op.

See: wc_OcspResponder_new

Return: void

This function releases all memory associated with an OcspResponder, including all CA entries, certificate statuses, private keys, and the internal RNG.

Requires HAVE_OCSP_RESPONDER to be defined.

C.37.2.12 function wc_OcspResponder_AddSigner

```
int wc_OcspResponder_AddSigner(
    OcspResponder * responder,
    const byte * signerDer,
    word32 signerDerSz,
    const byte * keyDer,
    word32 keyDerSz,
    const byte * issuerCertDer,
    word32 issuerCertDerSz
)
```

Adds a signing certificate and key to the OCSP Responder.

Parameters:

- **responder** Pointer to the OcspResponder.
- **signerDer** Pointer to the DER-encoded signer certificate.
- **signerDerSz** Size of the signer certificate in bytes.
- **keyDer** Pointer to the DER-encoded private key.
- **keyDerSz** Size of the private key in bytes.
- **issuerCertDer** Pointer to the DER-encoded issuing CA certificate, or NULL if the signer is itself the CA.
- **issuerCertDerSz** Size of the issuing CA certificate in bytes, or 0 if the signer is itself the CA.

See:

- wc_OcspResponder_new
- wc_OcspResponder_SetCertStatus
- wc_OcspResponder_WriteResponse

Return:

- 0 on success.
- BAD_FUNC_ARG if required parameters are NULL/zero, if the private key does not match the certificate, or if an authorized responder is missing the OCSP Signing extended key usage.
- MEMORY_E on memory allocation failure.
- DUPE_ENTRY_E if a signer with the same issuer hashes already exists.
- Other negative error codes on parsing or key loading failures.

This function registers a signer with the OCSP responder. The signer certificate and its corresponding private key are used to sign OCSP responses. The signer can be either:

- A CA certificate that directly issued the certificates being queried (issuerCertDer is NULL or zero-length).

- An authorized OCSP responder delegated by the issuing CA (issuerCertDer points to the issuing CA certificate). In this case the signer certificate must have the OCSP Signing extended key usage.

All certificate and key data must be in DER format.

Requires HAVE_OCSP_RESPONDER to be defined.

C.37.2.13 function wc_OcspResponder_SetCertStatus

```
int wc_OcspResponder_SetCertStatus(
    OcspResponder * responder,
    const char * caSubject,
    word32 caSubjectSz,
    const byte * serial,
    word32 serialSz,
    enum Ocsp_Cert_Status status,
    time_t revocationTime,
    enum WC_CRL_Reason revocationReason,
    word32 validityPeriod
)
```

Sets the revocation status of a certificate in the OCSP Responder.

Parameters:

- **responder** Pointer to the OcspResponder.
- **caSubject** The issuing CA subject name in the one-line distinguished name format used internally by the library (e.g. "/C=US/O=Org/CN=CA"). To avoid mismatches, obtain this value from wc_GetDecodedCertSubject() rather than constructing the string manually.
- **caSubjectSz** Length of the caSubject string in bytes, not including any NUL terminator.
- **serial** Pointer to the certificate serial number bytes.
- **serialSz** Size of the serial number in bytes.
- **status** Certificate status: CERT_GOOD, CERT_REVOKED, or CERT_UNKNOWN.
- **revocationTime** The time of revocation (only used when status is CERT_REVOKED, must be > 0 for revoked certs).
- **revocationReason** The CRL reason code for revocation (only used when status is CERT_REVOKED). See enum WC_CRL_Reason for valid values.
- **validityPeriod** Validity period in seconds for CERT_GOOD responses. Must be non-zero for CERT_GOOD status and zero for other statuses.

See:

- wc_OcspResponder_AddSigner
- wc_OcspResponder_WriteResponse

Return:

- 0 on success.
- BAD_FUNC_ARG if required parameters are NULL/zero or if status/revocation parameters are inconsistent.
- ASN_NO_SIGNER_E if no CA with the given subject name is found.
- MEMORY_E on memory allocation failure.

This function configures the status that the responder will report for a specific certificate identified by its serial number. The certificate must belong to a CA that has been previously added via wc_OcspResponder_AddSigner(). The CA is identified by its subject name.

If a status entry already exists for the given serial number, it will be updated.

Requires HAVE_OCSP_RESPONDER to be defined.

C.37.2.14 function wc_OcspResponder_WriteResponse

```
int wc_OcspResponder_WriteResponse(
    OcspResponder * responder,
    const byte * request,
    word32 requestSz,
    byte * response,
    word32 * responseSz
)
```

Generates an OCSP response for a given OCSP request.

Parameters:

- **responder** Pointer to the OcspResponder.
- **request** Pointer to the DER-encoded OCSP request.
- **requestSz** Size of the OCSP request in bytes.
- **response** Pointer to the output buffer for the DER-encoded OCSP response.
- **responseSz** Pointer to a word32 containing the size of the response buffer on input, and receiving the actual response size on output.

See:

- wc_OcspResponder_new
- wc_OcspResponder_AddSigner
- wc_OcspResponder_SetCertStatus
- wc_OcspResponder_WriteErrorResponse

Return:

- 0 on success.
- BAD_FUNC_ARG if required parameters are NULL/zero.
- Negative error codes on decode, lookup, or encoding failures.

This function processes a DER-encoded OCSP request, looks up the certificate status in the responder's database, and produces a signed OCSP response. The responder must have a signer and certificate status entries configured via wc_OcspResponder_AddSigner() and wc_OcspResponder_SetCertStatus() before calling this function.

Requires HAVE_OCSP_RESPONDER to be defined.

C.37.2.15 function wc_OcspResponder_WriteErrorResponse

```
int wc_OcspResponder_WriteErrorResponse(
    enum Ocsp_Response_Status status,
    byte * response,
    word32 * responseSz
)
```

Generates an OCSP error response.

Parameters:

- **status** The OCSP error status code. Must not be OCSP_SUCCESSFUL. Valid values include OCSP_MALFORMED_REQUEST, OCSP_INTERNAL_ERROR, OCSP_TRY_LATER, OCSP_SIG_REQUIRED, and OCSP_UNAUTHORIZED.
- **response** Pointer to the output buffer for the DER-encoded error response.
- **responseSz** Pointer to a word32 containing the size of the response buffer on input, and receiving the actual response size on output.

See: wc_OcspResponder_WriteResponse

Return:

- 0 on success.
- BAD_FUNC_ARG if responseSz is NULL or if status is OCSP_SUCCESSFUL.
- Negative error codes on encoding failure.

This function encodes an OCSP response with an error status code and no response body. This is used when the responder cannot process a request (e.g., malformed request, internal error, unauthorized).

Requires HAVE_OCSP_RESPONDER to be defined.

C.37.3 Source code

```
WOLFSSL_OCSP* wc_NewOCSP(WOLFSSL_CERT_MANAGER* cm);

void wc_FreeOCSP(WOLFSSL_OCSP* ocsp);

int wc_CheckCertOcspResponse(WOLFSSL_OCSP *ocsp, DecodedCert *cert, byte
 ↪  *response, int responseSz, void* heap);

OcspRequest* wc_OcspRequest_new(void* heap);

void wc_OcspRequest_free(OcspRequest* request);

int wc_InitOcspRequest(OcspRequest* req, DecodedCert* cert,
                                    byte useNonce, void* heap);

int wc_EncodeOcspRequest(OcspRequest* req, byte* output,
                                      word32 size);

OcspResponse* wc_OcspResponse_new(void* heap);

void wc_OcspResponse_free(OcspResponse* response);

OcspResponder* wc_OcspResponder_new(void* heap, int sendCerts);

void wc_OcspResponder_free(OcspResponder* responder);

int wc_OcspResponder_AddSigner(OcspResponder* responder,
    const byte* signerDer, word32 signerDerSz,
    const byte* keyDer, word32 keyDerSz,
    const byte* issuerCertDer, word32 issuerCertDerSz);

int wc_OcspResponder_SetCertStatus(OcspResponder* responder,
    const char* caSubject, word32 caSubjectSz,
    const byte* serial, word32 serialSz, enum Ocsp_Cert_Status status,
    time_t revocationTime, enum WC_CRL_Reason revocationReason,
    word32 validityPeriod);

int wc_OcspResponder_WriteResponse(OcspResponder* responder,
    const byte* request, word32 requestSz,
    byte* response, word32* responseSz);
```

```
int wc_OcspResponder_WriteErrorResponse(
    enum Ocsp_Response_Status status,
    byte* response, word32* responseSz);
```

C.38 pem.h

C.38.1 Functions

	Name
int	**wolfSSL_PEM_write_bio_PrivateKey**(WOLFSSL_BIO * bio, WOLFSSL_EVP_PKEY * key, const WOLFSSL_EVP_CIPHER * cipher, unsigned char * passwd, int len, wc_pem_password_cb * cb, void * arg)This function writes a key into a WOLFSSL_BIO structure in PEM format.

C.38.2 Functions Documentation

C.38.2.1 function wolfSSL_PEM_write_bio_PrivateKey

```
int wolfSSL_PEM_write_bio_PrivateKey(
    WOLFSSL_BIO * bio,
    WOLFSSL_EVP_PKEY * key,
    const WOLFSSL_EVP_CIPHER * cipher,
    unsigned char * passwd,
    int len,
    wc_pem_password_cb * cb,
    void * arg
)
```

This function writes a key into a WOLFSSL_BIO structure in PEM format.

Parameters:

- **bio** WOLFSSL_BIO structure to get PEM buffer from.
- **key** key to convert to PEM format.
- **cipher** EVP cipher structure.
- **passwd** password.
- **len** length of password.
- **cb** password callback.
- **arg** optional argument.

See: wolfSSL_PEM_read_bio_X509_AUX

Return:

- SSL_SUCCESS upon success.
- SSL_FAILURE upon failure.

Example

```
WOLFSSL_BIO* bio;
WOLFSSL_EVP_PKEY* key;
int ret;
// create bio and setup key
ret = wolfSSL_PEM_write_bio_PrivateKey(bio, key, NULL, NULL, 0, NULL, NULL);
//check ret value
```

C.38.3 Source code

```
int wolfSSL_PEM_write_bio_PrivateKey(WOLFSSL_BIO* bio, WOLFSSL_EVP_PKEY* key,
                                     const WOLFSSL_EVP_CIPHER* cipher,
                                     unsigned char* passwd, int len,
                                     wc_pem_password_cb* cb, void* arg);
```

C.39 pkcs11.h

C.39.1 Functions

	Name
int	**wc_Pkcs11_Initialize**(Pkcs11Dev * dev, const char * library, void * heap)
void	**wc_Pkcs11_Finalize**(Pkcs11Dev * dev)
int	**wc_Pkcs11Token_Init**(Pkcs11Token * token, Pkcs11Dev * dev, int slotId, const char * tokenName, const unsigned char * userPin, int userPinSz)
void	**wc_Pkcs11Token_Final**(Pkcs11Token * token)
int	**wc_Pkcs11Token_Open**(Pkcs11Token * token, int readWrite)
void	**wc_Pkcs11Token_Close**(Pkcs11Token * token)
int	**wc_Pkcs11StoreKey**(Pkcs11Token * token, int type, int clear, void * key)
int	**wc_Pkcs11_CryptoDevCb**(int devId, wc_CryptoInfo * info, void * ctx)

C.39.2 Functions Documentation

C.39.2.1 function wc_Pkcs11_Initialize

```
int wc_Pkcs11_Initialize(
    Pkcs11Dev * dev,
    const char * library,
    void * heap
)
```

C.39.2.2 function wc_Pkcs11_Finalize

```
void wc_Pkcs11_Finalize(
    Pkcs11Dev * dev
)
```

C.39.2.3 function wc_Pkcs11Token_Init

```
int wc_Pkcs11Token_Init(
    Pkcs11Token * token,
    Pkcs11Dev * dev,
    int slotId,
    const char * tokenName,
    const unsigned char * userPin,
```

```
    int userPinSz
)
```

C.39.2.4 function wc_Pkcs11Token_Final

```
void wc_Pkcs11Token_Final(
    Pkcs11Token * token
)
```

C.39.2.5 function wc_Pkcs11Token_Open

```
int wc_Pkcs11Token_Open(
    Pkcs11Token * token,
    int readWrite
)
```

C.39.2.6 function wc_Pkcs11Token_Close

```
void wc_Pkcs11Token_Close(
    Pkcs11Token * token
)
```

C.39.2.7 function wc_Pkcs11StoreKey

```
int wc_Pkcs11StoreKey(
    Pkcs11Token * token,
    int type,
    int clear,
    void * key
)
```

C.39.2.8 function wc_Pkcs11_CryptoDevCb

```
int wc_Pkcs11_CryptoDevCb(
    int devId,
    wc_CryptoInfo * info,
    void * ctx
)
```

C.39.3 Source code

```
int wc_Pkcs11_Initialize(Pkcs11Dev* dev, const char* library,
                                     void* heap);

void wc_Pkcs11_Finalize(Pkcs11Dev* dev);

int wc_Pkcs11Token_Init(Pkcs11Token* token, Pkcs11Dev* dev,
    int slotId, const char* tokenName, const unsigned char *userPin,
    int userPinSz);

void wc_Pkcs11Token_Final(Pkcs11Token* token);

int wc_Pkcs11Token_Open(Pkcs11Token* token, int readWrite);
```

```
void wc_Pkcs11Token_Close(Pkcs11Token* token);

int wc_Pkcs11StoreKey(Pkcs11Token* token, int type, int clear,
    void* key);

int wc_Pkcs11_CryptoDevCb(int devId, wc_CryptoInfo* info,
    void* ctx);
```

C.40 pkcs12.h

C.40.1 Functions

	Name
WC_PKCS12 *	**wc_PKCS12_new**(void)This function creates a new empty WC_PKCS12 structure, using the default heap hint for dynamic memory. It is equivalent to calling wc_PKCS12_new_ex() with a NULL heap. The returned structure must be freed with wc_PKCS12_free().
WC_PKCS12 *	**wc_PKCS12_new_ex**(void * heap)This function creates a new empty WC_PKCS12 structure, associating it with the heap hint given. The heap hint is stored in the structure and is used for every subsequent dynamic allocation and free made on behalf of this WC_PKCS12 object. The returned structure must be freed with wc_PKCS12_free().
void	**wc_PKCS12_free**(WC_PKCS12 * pkcs12)This function frees a WC_PKCS12 structure and all of the memory associated with it, including the parsed authenticated safe and the MAC data. Passing NULL is safe and does nothing.
int	**wc_d2i_PKCS12**(const byte * der, word32 derSz, WC_PKCS12 * pkcs12)This function converts a DER_encoded PKCS #12 (PFX) buffer into a WC_PKCS12 structure. The raw contents of each ContentInfo are stored in the structure without being completely parsed or decoded; call wc_PKCS12_parse() afterwards to decrypt the bundle and recover the private key and certificates.

	Name
int	**wc_d2i_PKCS12_fp**(const char * file, WC_PKCS12 ** pkcs12)This function reads a DER_encoded PKCS #12 (PFX) file from the file system and decodes it into a WC_PKCS12 structure. If *pkcs12 is NULL, a new WC_PKCS12 structure is allocated for the caller and returned through the pkcs12 argument; that structure is freed automatically if the decode fails. In either case, a successfully returned structure must be freed by the caller with wc_PKCS12_free(). This function is not available when NO_FILESYSTEM is defined.
int	**wc_i2d_PKCS12**(WC_PKCS12 * pkcs12, byte ** der, int * derSz)This function encodes a WC_PKCS12 structure into a DER_encoded PKCS #12 (PFX) buffer. It supports three modes of operation, selected by the der and derSz arguments. Passing NULL for der queries the required buffer size only: the size is stored in *derSz, LENGTH_ONLY_E is returned, and no data is written. Passing a non_NULL der whose *der is NULL allocates a buffer of the required size and stores its address in *der, which the caller must free with XFREE() using DYNAMIC_TYPE_PKCS. Passing a non-NULL der whose *der points to a caller-supplied buffer writes the DER into that buffer, and returns BUFFER_E if a non-NULL derSz indicates that the buffer is too small. In that last case *der is advanced on success to one byte past the end of the encoded DER, following the usual i2d convention, so the caller must keep its own copy of the original pointer.

	Name
int	**wc_PKCS12_parse**(WC_PKCS12 * pkcs12, const char * psw, byte ** pkey, word32 * pkeySz, byte ** cert, word32 * certSz, WC_DerCertList ** ca)This function parses and decodes a WC_PKCS12 structure that was previously populated by wc_d2i_PKCS12() or wc_d2i_PKCS12_fp(). The MAC on the bundle is verified with the given password, the contents are decrypted, and the private key, the end entity certificate, and optionally the CA certificate chain are returned to the caller. The key and certificate are returned in newly allocated buffers which the caller is responsible for freeing: use XFREE(pkey, heap, DYNAMIC_TYPE_PUBLIC_KEY) for the key and XFREE(cert, heap, DYNAMIC_TYPE_PKCS) for the certificate, where heap is the heap hint associated with the WC_PKCS12 structure. The CA list, when requested, must be freed with wc_FreeCertList(). The private key is returned with the PKCS #8 header removed; use wc_PKCS12_parse_ex() if the PKCS #8 header should be kept.
int	**wc_PKCS12_parse_ex**(WC_PKCS12 * pkcs12, const char * psw, byte ** pkey, word32 * pkeySz, byte ** cert, word32 * certSz, WC_DerCertList ** ca, int keepKeyHeader)This function behaves exactly like wc_PKCS12_parse() but adds the keepKeyHeader argument, which controls whether the PKCS #8 header is left on the returned private key. Calling wc_PKCS12_parse() is equivalent to calling this function with keepKeyHeader set to 0. The same ownership rules as wc_PKCS12_parse() apply: free the key with XFREE(pkey, heap, DYNAMIC_TYPE_PUBLIC_KEY), the certificate with XFREE(cert, heap, DYNAMIC_TYPE_PKCS), and the CA list with wc_FreeCertList().

COPYRIGHT ©2024 wolfSSL Inc.

	Name
WC_PKCS12 *	**wc_PKCS12_create**(char * pass, word32 passSz, char * name, byte * key, word32 keySz, byte * cert, word32 certSz, WC_DerCertList * ca, int nidKey, int nidCert, int iter, int macIter, int keyType, void * heap)This function creates a new WC_PKCS12 structure from a DER_encoded private key, a DER_encoded certificate, and an optional list of extra certificates. The key and certificate are each placed in their own ContentInfo and optionally encrypted with the password given. A MAC is computed over the result only when macIter is greater than zero. The returned structure can be encoded to DER with wc_i2d_PKCS12() and must be freed with wc_PKCS12_free(). The nidKey and nidCert arguments select the password-based encryption applied to the key and to the certificate respectively. Supported values are PBE_SHA1_RC4_128, PBE_SHA1_DES, PBE_SHA1_DES3, PBE_AES128_CBC and PBE_AES256_CBC. Passing -1 stores the corresponding content unencrypted.
void	**wc_FreeCertList**(WC_DerCertList * list, void * heap)This function frees a WC_DerCertList linked list, including the DER buffer held by each node. It is used to release the CA certificate list returned by wc_PKCS12_parse() and wc_PKCS12_parse_ex(). The heap hint given must match the one associated with the WC_PKCS12 structure the list came from. Passing NULL for the list is safe and does nothing.

C.40.2 Functions Documentation

C.40.2.1 function wc_PKCS12_new

```
WC_PKCS12 * wc_PKCS12_new(
    void
)
```

This function creates a new empty WC_PKCS12 structure, using the default heap hint for dynamic memory. It is equivalent to calling wc_PKCS12_new_ex() with a NULL heap. The returned structure must be freed with wc_PKCS12_free().

Parameters:

- **none** No parameters.

See:

- wc_PKCS12_new_ex
- wc_PKCS12_free

Return:

- pointer Returns a pointer to a newly allocated WC_PKCS12 structure on success
- NULL Returned if the allocation fails

Example

```
WC_PKCS12* pkcs12 = wc_PKCS12_new();
if (pkcs12 == NULL) {
    // error allocating PKCS12 structure
}

// use the PKCS12 structure

wc_PKCS12_free(pkcs12);
```

C.40.2.2 function wc_PKCS12_new_ex

```
WC_PKCS12 * wc_PKCS12_new_ex(
    void * heap
)
```

This function creates a new empty WC_PKCS12 structure, associating it with the heap hint given. The heap hint is stored in the structure and is used for every subsequent dynamic allocation and free made on behalf of this WC_PKCS12 object. The returned structure must be freed with wc_PKCS12_free().

Parameters:

- **heap** pointer to a heap hint used for dynamic memory allocation, or NULL to use the default

See:

- wc_PKCS12_new
- wc_PKCS12_free

Return:

- pointer Returns a pointer to a newly allocated WC_PKCS12 structure on success
- NULL Returned if the allocation fails

Example

```
void* heap = NULL; // or a custom static memory heap hint
WC_PKCS12* pkcs12 = wc_PKCS12_new_ex(heap);
if (pkcs12 == NULL) {
    // error allocating PKCS12 structure
}

// use the PKCS12 structure

wc_PKCS12_free(pkcs12);
```

C.40.2.3 function wc_PKCS12_free

```
void wc_PKCS12_free(
    WC_PKCS12 * pkcs12
)
```

This function frees a WC_PKCS12 structure and all of the memory associated with it, including the parsed authenticated safe and the MAC data. Passing NULL is safe and does nothing.

Parameters:

- **pkcs12** pointer to the WC_PKCS12 structure to free

See:

- wc_PKCS12_new
- wc_PKCS12_new_ex

Return: none No returns.

Example

```
WC_PKCS12* pkcs12 = wc_PKCS12_new();

// initialize and use the PKCS12 structure

wc_PKCS12_free(pkcs12);
```

C.40.2.4 function wc_d2i_PKCS12

```
int wc_d2i_PKCS12(
    const byte * der,
    word32 derSz,
    WC_PKCS12 * pkcs12
)
```

This function converts a DER-encoded PKCS #12 (PFX) buffer into a WC_PKCS12 structure. The raw contents of each ContentInfo are stored in the structure without being completely parsed or decoded; call wc_PKCS12_parse() afterwards to decrypt the bundle and recover the private key and certificates.

Parameters:

- **der** pointer to a buffer holding the DER-encoded PKCS #12 bundle
- **derSz** size of the DER buffer
- **pkcs12** pointer to an allocated WC_PKCS12 structure in which to store the decoded bundle

See:

- wc_d2i_PKCS12_fp
- wc_i2d_PKCS12
- wc_PKCS12_parse

Return:

- 0 Returned on successfully decoding the PKCS #12 buffer
- BAD_FUNC_ARG Returned if der or pkcs12 is NULL
- ASN_PARSE_E Returned if there is an error parsing the PKCS #12 structure
- ASN_VERSION_E Returned if the version in the bundle is not supported
- MEMORY_E Returned if there is an error allocating memory

Example

```
WC_PKCS12* pkcs12 = NULL;
byte der[] = { }; // initialize with a DER-encoded PKCS #12 bundle
word32 derSz = sizeof(der);

pkcs12 = wc_PKCS12_new();
if (pkcs12 == NULL) {
    // error allocating PKCS12 structure
}

if (wc_d2i_PKCS12(der, derSz, pkcs12) != 0) {
```

```
    // error decoding PKCS12 bundle
}

wc_PKCS12_free(pkcs12);
```

C.40.2.5 function wc_d2i_PKCS12_fp

```
int wc_d2i_PKCS12_fp(
    const char * file,
    WC_PKCS12 ** pkcs12
)
```

This function reads a DER-encoded PKCS #12 (PFX) file from the file system and decodes it into a WC_PKCS12 structure. If *pkcs12 is NULL, a new WC_PKCS12 structure is allocated for the caller and returned through the pkcs12 argument; that structure is freed automatically if the decode fails. In either case, a successfully returned structure must be freed by the caller with wc_PKCS12_free(). This function is not available when NO_FILESYSTEM is defined.

Parameters:

- **file** path of the DER-encoded PKCS #12 file to read
- **pkcs12** pointer to a WC_PKCS12 pointer. If *pkcs12 is NULL, a new structure is allocated and returned here; otherwise the existing structure is used.

See:

- wc_d2i_PKCS12
- wc_i2d_PKCS12
- wc_PKCS12_parse

Return:

- 0 Returned on successfully reading and decoding the file
- BAD_FUNC_ARG Returned if pkcs12 is NULL
- MEMORY_E Returned if there is an error allocating memory
- BAD_PATH_ERROR Returned if the file cannot be opened or read

Example

```
WC_PKCS12* pkcs12 = NULL;

// let wc_d2i_PKCS12_fp allocate the structure for us
if (wc_d2i_PKCS12_fp("./certs/test-servercert.p12", &pkcs12) != 0) {
    // error reading or decoding the PKCS12 file
}

wc_PKCS12_free(pkcs12);
```

C.40.2.6 function wc_i2d_PKCS12

```
int wc_i2d_PKCS12(
    WC_PKCS12 * pkcs12,
    byte ** der,
    int * derSz
)
```

This function encodes a WC_PKCS12 structure into a DER-encoded PKCS #12 (PFX) buffer. It supports three modes of operation, selected by the der and derSz arguments. Passing NULL for der queries the required buffer size only: the size is stored in *derSz, LENGTH_ONLY_E is returned, and no data is

written. Passing a non-NULL der whose *der is NULL allocates a buffer of the required size and stores its address in *der, which the caller must free with XFREE() using DYNAMIC_TYPE_PKCS. Passing a non-NULL der whose *der points to a caller-supplied buffer writes the DER into that buffer, and returns BUFFER_E if a non-NULL derSz indicates that the buffer is too small. In that last case *der is advanced on success to one byte past the end of the encoded DER, following the usual i2d convention, so the caller must keep its own copy of the original pointer.

Parameters:

- **pkcs12** pointer to the WC_PKCS12 structure to encode
- **der** pointer to a buffer pointer in which to store the encoding, or NULL to query the required size
- **derSz** on a size query, receives the required size. When a caller-supplied buffer is used, holds the size of that buffer on input; a NULL value disables the buffer size check.

See:

- wc_d2i_PKCS12
- wc_PKCS12_create

Return:

- Success On success, returns the size of the DER encoding in bytes
- LENGTH_ONLY_E Returned when der is NULL, indicating that only the required size was computed and stored in *derSz
- BAD_FUNC_ARG Returned if pkcs12 is NULL, if the structure holds no authenticated safe, or if both der and derSz are NULL
- BUFFER_E Returned if a caller-supplied buffer is too small to hold the encoding
- MEMORY_E Returned if there is an error allocating memory

Example

```
WC_PKCS12* pkcs12 = NULL;
byte* der = NULL;
int derSz = 0;

// pkcs12 previously created with wc_PKCS12_create or decoded with
// wc_d2i_PKCS12

// query the required size
if (wc_i2d_PKCS12(pkcs12, NULL, &derSz) != LENGTH_ONLY_E) {
    // error getting the encoded size
}

// let wc_i2d_PKCS12 allocate the buffer for us
if (wc_i2d_PKCS12(pkcs12, &der, NULL) <= 0) {
    // error encoding the PKCS12 bundle
}

XFREE(der, NULL, DYNAMIC_TYPE_PKCS);
wc_PKCS12_free(pkcs12);
```

C.40.2.7 function wc_PKCS12_parse

```
int wc_PKCS12_parse(
    WC_PKCS12 * pkcs12,
    const char * psw,
    byte ** pkey,
    word32 * pkeySz,
```

```
    byte ** cert,
    word32 * certSz,
    WC_DerCertList ** ca
)
```

This function parses and decodes a WC_PKCS12 structure that was previously populated by wc_d2i_PKCS12() or wc_d2i_PKCS12_fp(). The MAC on the bundle is verified with the given password, the contents are decrypted, and the private key, the end entity certificate, and optionally the CA certificate chain are returned to the caller. The key and certificate are returned in newly allocated buffers which the caller is responsible for freeing: use XFREE(pkey, heap, DYNAMIC_TYPE_PUBLIC_KEY) for the key and XFREE(cert, heap, DYNAMIC_TYPE_PKCS) for the certificate, where heap is the heap hint associated with the WC_PKCS12 structure. The CA list, when requested, must be freed with wc_FreeCertList(). The private key is returned with the PKCS #8 header removed; use wc_PKCS12_parse_ex() if the PKCS #8 header should be kept.

Parameters:

- **pkcs12** pointer to a WC_PKCS12 structure holding a decoded bundle
- **psw** NULL-terminated password used to verify the MAC and decrypt the bundle
- **pkey** receives a newly allocated buffer holding the DER-encoded private key
- **pkeySz** receives the size of the private key buffer
- **cert** receives a newly allocated buffer holding the DER-encoded certificate
- **certSz** receives the size of the certificate buffer
- **ca** optional. If non-NULL, receives a linked list of the remaining DER-encoded certificates found in the bundle. Free with wc_FreeCertList().

See:

- wc_PKCS12_parse_ex
- wc_d2i_PKCS12
- wc_FreeCertList

Return:

- 0 Returned on successfully parsing the PKCS #12 bundle
- BAD_FUNC_ARG Returned if pkcs12, psw, pkey, pkeySz, cert or certSz is NULL
- MAC_CMP_FAILED_E Returned if the MAC verification fails, which usually means the password is incorrect
- ASN_PARSE_E Returned if there is an error parsing the contents
- MEMORY_E Returned if there is an error allocating memory
- UNICODE_SIZE_E Returned if the password cannot be converted to the Unicode form required for key derivation

Note: When USER_RSA is enabled this function may return a certificate that is not the pair of the returned key when RSA key pairs are used.

Example

```
WC_PKCS12* pkcs12 = NULL;
byte* key = NULL;
byte* cert = NULL;
WC_DerCertList* ca = NULL;
word32 keySz = 0;
word32 certSz = 0;

if (wc_d2i_PKCS12_fp("./certs/test-servercert.p12", &pkcs12) != 0) {
    // error reading the PKCS12 file
}
```

```
if (wc_PKCS12_parse(pkcs12, "password", &key, &keySz, &cert, &certSz,
                    &ca) != 0) {
    // error parsing the bundle, e.g. wrong password
}

// use the key, cert and ca chain

XFREE(key, NULL, DYNAMIC_TYPE_PUBLIC_KEY);
XFREE(cert, NULL, DYNAMIC_TYPE_PKCS);
wc_FreeCertList(ca, NULL);
wc_PKCS12_free(pkcs12);
```

C.40.2.8 function wc_PKCS12_parse_ex

```
int wc_PKCS12_parse_ex(
    WC_PKCS12 * pkcs12,
    const char * psw,
    byte ** pkey,
    word32 * pkeySz,
    byte ** cert,
    word32 * certSz,
    WC_DerCertList ** ca,
    int keepKeyHeader
)
```

This function behaves exactly like wc_PKCS12_parse() but adds the keepKeyHeader argument, which controls whether the PKCS #8 header is left on the returned private key. Calling wc_PKCS12_parse() is equivalent to calling this function with keepKeyHeader set to 0. The same ownership rules as wc_PKCS12_parse() apply: free the key with XFREE(pkey, heap, DYNAMIC_TYPE_PUBLIC_KEY), the certificate with XFREE(cert, heap, DYNAMIC_TYPE_PKCS), and the CA list with wc_FreeCertList().

Parameters:

- **pkcs12** pointer to a WC_PKCS12 structure holding a decoded bundle
- **psw** NULL-terminated password used to verify the MAC and decrypt the bundle
- **pkey** receives a newly allocated buffer holding the DER-encoded private key
- **pkeySz** receives the size of the private key buffer
- **cert** receives a newly allocated buffer holding the DER-encoded certificate
- **certSz** receives the size of the certificate buffer
- **ca** optional. If non-NULL, receives a linked list of the remaining DER-encoded certificates found in the bundle. Free with wc_FreeCertList().
- **keepKeyHeader** 0 to strip the PKCS #8 header from the returned key, any other value to keep it

See:

- wc_PKCS12_parse
- wc_d2i_PKCS12
- wc_FreeCertList

Return:

- 0 Returned on successfully parsing the PKCS #12 bundle
- BAD_FUNC_ARG Returned if pkcs12, psw, pkey, pkeySz, cert or certSz is NULL
- MAC_CMP_FAILED_E Returned if the MAC verification fails, which usually means the password is incorrect
- ASN_PARSE_E Returned if there is an error parsing the contents
- MEMORY_E Returned if there is an error allocating memory

- UNICODE_SIZE_E Returned if the password cannot be converted to the Unicode form required for key derivation

Note: When USER_RSA is enabled this function may return a certificate that is not the pair of the returned key when RSA key pairs are used.

Example

```
WC_PKCS12* pkcs12 = NULL;
byte* key = NULL;
byte* cert = NULL;
WC_DerCertList* ca = NULL;
word32 keySz = 0;
word32 certSz = 0;

// pkcs12 previously decoded with wc_d2i_PKCS12 or wc_d2i_PKCS12_fp

// keep the PKCS #8 header on the returned private key
if (wc_PKCS12_parse_ex(pkcs12, "password", &key, &keySz, &cert, &certSz,
                       &ca, 1) != 0) {
    // error parsing the bundle
}

XFREE(key, NULL, DYNAMIC_TYPE_PUBLIC_KEY);
XFREE(cert, NULL, DYNAMIC_TYPE_PKCS);
wc_FreeCertList(ca, NULL);
wc_PKCS12_free(pkcs12);
```

C.40.2.9 function wc_PKCS12_create

```
WC_PKCS12 * wc_PKCS12_create(
    char * pass,
    word32 passSz,
    char * name,
    byte * key,
    word32 keySz,
    byte * cert,
    word32 certSz,
    WC_DerCertList * ca,
    int nidKey,
    int nidCert,
    int iter,
    int macIter,
    int keyType,
    void * heap
)
```

This function creates a new WC_PKCS12 structure from a DER-encoded private key, a DER-encoded certificate, and an optional list of extra certificates. The key and certificate are each placed in their own ContentInfo and optionally encrypted with the password given. A MAC is computed over the result only when macIter is greater than zero. The returned structure can be encoded to DER with wc_i2d_PKCS12() and must be freed with wc_PKCS12_free(). The nidKey and nidCert arguments select the password-based encryption applied to the key and to the certificate respectively. Supported values are PBE_SHA1_RC4_128, PBE_SHA1_DES, PBE_SHA1_DES3, PBE_AES128_CBC and PBE_AES256_CBC. Passing -1 stores the corresponding content unencrypted.

Parameters:

- **pass** password to use for encryption and for the MAC
- **passSz** size of the password buffer
- **name** friendlyName to use. Not currently used.
- **key** buffer holding the DER-encoded private key
- **keySz** size of the key buffer
- **cert** buffer holding the DER-encoded certificate
- **certSz** size of the certificate buffer
- **ca** optional linked list of additional DER-encoded certificates to include, or NULL
- **nidKey** encryption to apply to the private key, or -1 for none
- **nidCert** encryption to apply to the certificate, or -1 for none
- **iter** number of iterations to use for the encryption. Values of 0 or less select WC_PKCS12_ITT_DEFAULT.
- **macIter** number of iterations to use when creating the MAC. A MAC is only created when this is greater than zero. Unlike iter, a value of zero or less does not select a default: no MAC is created at all, and the resulting bundle has no MAC integrity protection.
- **keyType** flag for a signature and/or encryption key. Not currently used.
- **heap** pointer to a heap hint used for dynamic memory allocation, or NULL to use the default

See:

- wc_i2d_PKCS12
- wc_PKCS12_parse
- wc_PKCS12_free

Return:

- pointer Returns a pointer to a newly created WC_PKCS12 structure on success
- NULL Returned if the RNG cannot be initialized, if memory allocation fails, if an unsupported nidKey or nidCert is given, or if the bundle cannot be built

Note: The name and keyType arguments are accepted for API compatibility but are not currently used.

Example

```
WC_PKCS12* pkcs12 = NULL;
byte* der = NULL;
char pass[] = "password";
byte key[] = { };  // initialize with a DER-encoded private key
byte cert[] = { }; // initialize with a DER-encoded certificate

pkcs12 = wc_PKCS12_create(pass, sizeof(pass) - 1, NULL,
                          key, sizeof(key), cert, sizeof(cert), NULL,
                          PBE_AES256_CBC, PBE_AES256_CBC,
                          WC_PKCS12_ITT_DEFAULT, WC_PKCS12_ITT_DEFAULT,
                          0, NULL);
if (pkcs12 == NULL) {
    // error creating the PKCS12 bundle
}

if (wc_i2d_PKCS12(pkcs12, &der, NULL) <= 0) {
    // error encoding the bundle to DER
}

XFREE(der, NULL, DYNAMIC_TYPE_PKCS);
wc_PKCS12_free(pkcs12);
```

C.40.2.10 function wc_FreeCertList

```
void wc_FreeCertList(
    WC_DerCertList * list,
    void * heap
)
```

This function frees a WC_DerCertList linked list, including the DER buffer held by each node. It is used to release the CA certificate list returned by wc_PKCS12_parse() and wc_PKCS12_parse_ex(). The heap hint given must match the one associated with the WC_PKCS12 structure the list came from. Passing NULL for the list is safe and does nothing.

Parameters:

- **list** pointer to the head of the WC_DerCertList to free
- **heap** pointer to the heap hint used when the list was allocated

See:

- wc_PKCS12_parse
- wc_PKCS12_parse_ex

Return: none No returns.

Example

```
WC_DerCertList* ca = NULL;

// ca populated by a previous call to wc_PKCS12_parse

wc_FreeCertList(ca, NULL);
```

C.40.3 Source code

```
WC_PKCS12* wc_PKCS12_new(void);

WC_PKCS12* wc_PKCS12_new_ex(void* heap);

void wc_PKCS12_free(WC_PKCS12* pkcs12);

int wc_d2i_PKCS12(const byte* der, word32 derSz, WC_PKCS12* pkcs12);

int wc_d2i_PKCS12_fp(const char* file, WC_PKCS12** pkcs12);

int wc_i2d_PKCS12(WC_PKCS12* pkcs12, byte** der, int* derSz);

int wc_PKCS12_parse(WC_PKCS12* pkcs12, const char* psw,
        byte** pkey, word32* pkeySz, byte** cert, word32* certSz,
        WC_DerCertList** ca);

int wc_PKCS12_parse_ex(WC_PKCS12* pkcs12, const char* psw,
        byte** pkey, word32* pkeySz, byte** cert, word32* certSz,
        WC_DerCertList** ca, int keepKeyHeader);

WC_PKCS12* wc_PKCS12_create(char* pass, word32 passSz,
        char* name, byte* key, word32 keySz, byte* cert, word32 certSz,
        WC_DerCertList* ca, int nidKey, int nidCert, int iter, int macIter,
        int keyType, void* heap);
```

```
void wc_FreeCertList(WC_DerCertList* list, void* heap);
```

C.41 pkcs7.h

C.41.1 Types

	Name
typedef int()(*const byte* key, word32 keySz, const byte *in, word32 inSz, int wrap, byte* out, word32 outSz)	**CallbackAESKeyWrapUnwrap** Callback used for a custom AES key wrap/unwrap operation.

C.41.2 Functions

	Name
int	**wc_PKCS7_InitWithCert**(wc_PKCS7 * pkcs7, byte * der, word32 derSz)This function initializes a PKCS7 structure with a DER-formatted certificate. To initialize an empty PKCS7 structure, one can pass in a NULL cert and 0 for certSz.
void	**wc_PKCS7_Free**(wc_PKCS7 * pkcs7)This function releases any memory allocated by a PKCS7 initializer.
int	**wc_PKCS7_EncodeData**(wc_PKCS7 * pkcs7, byte * output, word32 outputSz)This function builds the PKCS7 data content type, encoding the PKCS7 structure into a buffer containing a parsable PKCS7 data packet.
int	**wc_PKCS7_EncodeSignedData**(wc_PKCS7 * pkcs7, byte * output, word32 outputSz)This function builds the PKCS7 signed data content type, encoding the PKCS7 structure into a buffer containing a parsable PKCS7 signed data packet. For RSA_PSS signers (WC_RSA_PSS), see PKCS#7 RSA_PSS (CMS).
int	**wc_PKCS7_EncodeSignedData_ex**(wc_PKCS7 * pkcs7, const byte * hashBuf, word32 hashSz, byte * outputHead, word32 * outputHeadSz, byte * outputFoot, word32 * outputFootSz)This function builds the PKCS7 signed data content type, encoding the PKCS7 structure into a header and footer buffer containing a parsable PKCS7 signed data packet. This does not include the content. A hash must be computed and provided for the data.

	Name
int	**wc_PKCS7_VerifySignedData**(wc_PKCS7 * pkcs7, byte * pkiMsg, word32 pkiMsgSz)This function takes in a transmitted PKCS7 signed data message, extracts the certificate list and certificate revocation list, and then verifies the signature. It stores the extracted content in the given PKCS7 structure.
int	**wc_PKCS7_VerifySignedData_ex**(wc_PKCS7 * pkcs7, const byte * hashBuf, word32 hashSz, byte * pkiMsgHead, word32 pkiMsgHeadSz, byte * pkiMsgFoot, word32 pkiMsgFootSz)This function takes in a transmitted PKCS7 signed data message as hash/header/footer, then extracts the certificate list and certificate revocation list, and then verifies the signature. It stores the extracted content in the given PKCS7 structure.
int	**wc_PKCS7_SetAESKeyWrapUnwrapCb**(wc_PKCS7 * pkcs7, CallbackAESKeyWrapUnwrap aesKeyWrapCb)Set the callback function to be used to perform a custom AES key wrap/unwrap operation.
int	**wc_PKCS7_EncodeEnvelopedData**(wc_PKCS7 * pkcs7, byte * output, word32 outputSz)This function builds the PKCS7 enveloped data content type, encoding the PKCS7 structure into a buffer containing a parsable PKCS7 enveloped data packet.
int	**wc_PKCS7_DecodeEnvelopedData**(wc_PKCS7 * pkcs7, byte * pkiMsg, word32 pkiMsgSz, byte * output, word32 outputSz)This function unwraps and decrypts a PKCS7 enveloped data content type, decoding the message into output. It uses the private key of the PKCS7 object passed in to decrypt the message.
int	**wc_PKCS7_GetEnvelopedDataKariRid**(const byte * in, word32 inSz, byte * out, word32 * outSz)This function extracts the KeyAgreeRecipientIdentifier object from an EnvelopedData package containing a KeyAgreeRecipientInfo RecipientInfo object. Only the first KeyAgreeRecipientIdentifer found in the first RecipientInfo is copied. This function does not support multiple RecipientInfo objects or multiple RecipientEncryptedKey objects within an KeyAgreeRecipientInfo.

	Name
int	**wc_PKCS7_DecodeEncryptedData**(wc_PKCS7 * pkcs7, byte * pkiMsg, word32 pkiMsgSz, byte * output, word32 outputSz)This function unwraps and decrypts a PKCS7 encrypted data content type, decoding the message into output. It uses the encryption key of the PKCS7 object passed in via pkcs7->encryptionKey and pkcs7->encryptionKeySz to decrypt the message.
int	**wc_PKCS7_DecodeEncryptedKeyPackage**(wc_PKCS7 * pkcs7, byte * pkiMsg, word32 pkiMsgSz, byte * output, word32 outputSz)This function unwraps and decrypts a PKCS7 encrypted key package content type, decoding the message into output. If the wrapped content type is EncryptedData, the encryption key must be set in the pkcs7 input structure (via pkcs7_>encryptionKey and pkcs7_>encryptionKeySz). If the wrapped content type is EnvelopedData, the private key must be set in the pkcs7 input structure (via pkcs7_>privateKey and pkcs7_>privateKeySz). A wrapped content type of AuthEnvelopedData is not currently supported.
int	**wc_PKCS7_DecodeSymmetricKeyPackageAttribute**(const byte * skp, word32 skpSz, size_t index, const byte ** attr, word32 * attrSz)This function provides access to a SymmetricKeyPackage attribute.
int	**wc_PKCS7_DecodeSymmetricKeyPackageKey**(const byte * skp, word32 skpSz, size_t index, const byte ** key, word32 * keySz)This function provides access to a SymmetricKeyPackage key.
int	**wc_PKCS7_DecodeOneSymmetricKeyAttribute**(const byte * osk, word32 oskSz, size_t index, const byte ** attr, word32 * attrSz)This function provides access to a OneSymmetricKey attribute.
int	**wc_PKCS7_DecodeOneSymmetricKeyKey**(const byte * osk, word32 oskSz, const byte ** key, word32 * keySz)This function provides access to a OneSymmetricKey key.
PKCS7 *	**wolfSSL_PKCS7_new**(void)Creates new PKCS7 structure.
PKCS7_SIGNED *	**wolfSSL_PKCS7_SIGNED_new**(void)Creates new PKCS7_SIGNED structure.
void	**wolfSSL_PKCS7_free**(PKCS7 * p7)Frees PKCS7 structure.
void	**wolfSSL_PKCS7_SIGNED_free**(PKCS7_SIGNED * p7)Frees PKCS7_SIGNED structure.

	Name
PKCS7 *	**wolfSSL_d2i_PKCS7**(PKCS7 ** p7, const unsigned char ** in, int len)Decodes DER-encoded PKCS7 structure.
PKCS7 *	**wolfSSL_d2i_PKCS7_bio**(WOLFSSL_BIO * bio, PKCS7 ** p7)Decodes PKCS7 from BIO.
int	**wolfSSL_i2d_PKCS7_bio**(WOLFSSL_BIO * bio, PKCS7 * p7)Encodes PKCS7 to BIO.
int	**wolfSSL_i2d_PKCS7**(PKCS7 * p7, unsigned char ** out)Encodes PKCS7 to DER.
PKCS7 *	**wolfSSL_PKCS7_sign**(WOLFSSL_X509 * signer, WOLFSSL_EVP_PKEY * pkey, WOLFSSL_STACK * certs, WOLFSSL_BIO * in, int flags)Creates signed PKCS7 message.
int	**wolfSSL_PKCS7_verify**(PKCS7 * p7, WOLFSSL_STACK * certs, WOLFSSL_X509_STORE * store, WOLFSSL_BIO * in, WOLFSSL_BIO * out, int flags)Verifies signed PKCS7 message.
int	**wolfSSL_PKCS7_final**(PKCS7 * pkcs7, WOLFSSL_BIO * in, int flags)Finalizes PKCS7 structure with data.
int	**wolfSSL_PKCS7_encode_certs**(PKCS7 * p7, WOLFSSL_STACK * certs, WOLFSSL_BIO * out)Encodes certificates into PKCS7.
WOLFSSL_STACK *	**wolfSSL_PKCS7_to_stack**(PKCS7 * pkcs7)Converts PKCS7 certificates to stack.
WOLFSSL_STACK *	**wolfSSL_PKCS7_get0_signers**(PKCS7 * p7, WOLFSSL_STACK * certs, int flags)Gets signer certificates from PKCS7.
int	**wolfSSL_PEM_write_bio_PKCS7**(WOLFSSL_BIO * bio, PKCS7 * p7)Writes PKCS7 to BIO in PEM format.
PKCS7 *	**wolfSSL_SMIME_read_PKCS7**(WOLFSSL_BIO * in, WOLFSSL_BIO ** bcont)Reads S/MIME PKCS7 from BIO.
int	**wolfSSL_SMIME_write_PKCS7**(WOLFSSL_BIO * out, PKCS7 * pkcs7, WOLFSSL_BIO * in, int flags)Writes PKCS7 to BIO in S/MIME format.
wc_PKCS7 *	**wc_PKCS7_New**(void * heap, int devId)Creates new wc_PKCS7 structure.
void	**wc_PKCS7_SetUnknownExtCallback**(wc_PKCS7 * pkcs7, wc_UnknownExtCallback cb)Sets unknown extension callback.
int	**wc_PKCS7_Init**(wc_PKCS7 * pkcs7, void * heap, int devId)Initializes wc_PKCS7 structure.
int	**wc_PKCS7_AddCertificate**(wc_PKCS7 * pkcs7, byte * der, word32 derSz)Adds certificate to PKCS7.
int	**wc_PKCS7_GetAttributeValue**(wc_PKCS7 * pkcs7, const byte * oid, word32 oidSz, byte * out, word32 * outSz)Gets attribute value from PKCS7.

	Name
int	**wc_PKCS7_SetSignerIdentifierType**(wc_PKCS7 * pkcs7, int type)Sets signer identifier type.
int	**wc_PKCS7_SetContentType**(wc_PKCS7 * pkcs7, byte * contentType, word32 sz)Sets content type.
int	**wc_PKCS7_GetPadSize**(word32 inputSz, word32 blockSz)Gets padding size for block cipher.
int	**wc_PKCS7_PadData**(byte * in, word32 inSz, byte * out, word32 outSz, word32 blockSz)Pads data for block cipher.
int	**wc_PKCS7_SetCustomSKID**(wc_PKCS7 * pkcs7, const byte * in, word16 inSz)Sets custom subject key identifier.
int	**wc_PKCS7_SetDetached**(wc_PKCS7 * pkcs7, word16 flag)Sets detached signature flag.
int	**wc_PKCS7_NoDefaultSignedAttribs**(wc_PKCS7 * pkcs7)Disables default signed attributes.
int	**wc_PKCS7_SetDefaultSignedAttribs**(wc_PKCS7 * pkcs7, word16 flag)Sets default signed attributes flag.
void	**wc_PKCS7_AllowDegenerate**(wc_PKCS7 * pkcs7, word16 flag)Allows degenerate PKCS7 (no signers).
int	**wc_PKCS7_GetSignerSID**(wc_PKCS7 * pkcs7, byte * out, word32 * outSz)Gets signer subject identifier.
int	**wc_PKCS7_EncodeSignedFPD**(wc_PKCS7 * pkcs7, byte * privateKey, word32 privateKeySz, int signOID, int hashOID, byte * content, word32 contentSz, PKCS7Attrib * signedAttribs, word32 signedAttribsSz, byte * output, word32 outputSz)Encodes signed FirmwarePackageData.
int	**wc_PKCS7_EncodeSignedEncryptedFPD**(wc_PKCS7 * pkcs7, byte * encryptKey, word32 encryptKeySz, byte * privateKey, word32 privateKeySz, int encryptOID, int signOID, int hashOID, byte * content, word32 contentSz, PKCS7Attrib * unprotectedAttribs, word32 unprotectedAttribsSz, PKCS7Attrib * signedAttribs, word32 signedAttribsSz, byte * output, word32 outputSz)Encodes signed encrypted FirmwarePackageData.
int	**wc_PKCS7_EncodeSignedCompressedFPD**(wc_PKCS7 * pkcs7, byte * privateKey, word32 privateKeySz, int signOID, int hashOID, byte * content, word32 contentSz, PKCS7Attrib * signedAttribs, word32 signedAttribsSz, byte * output, word32 outputSz)Encodes signed compressed FirmwarePackageData.

	Name
int	**wc_PKCS7_EncodeSignedEncryptedCompressedFPD**(wc_PKCS7 * pkcs7, byte * encryptKey, word32 encryptKeySz, byte * privateKey, word32 privateKeySz, int encryptOID, int signOID, int hashOID, byte * content, word32 contentSz, PKCS7Attrib * unprotectedAttribs, word32 unprotectedAttribsSz, PKCS7Attrib * signedAttribs, word32 signedAttribsSz, byte * output, word32 outputSz)Encodes signed encrypted compressed FirmwarePackageData.
int	**wc_PKCS7_AddRecipient_KTRI**(wc_PKCS7 * pkcs7, const byte * cert, word32 certSz, int options)Adds KTRI recipient.
int	**wc_PKCS7_AddRecipient_KARI**(wc_PKCS7 * pkcs7, const byte * cert, word32 certSz, int keyWrapOID, int keyAgreeOID, byte * ukm, word32 ukmSz, int options)Adds KARI recipient.
int	**wc_PKCS7_SetKey**(wc_PKCS7 * pkcs7, byte * key, word32 keySz)Sets encryption key.
int	**wc_PKCS7_AddRecipient_KEKRI**(wc_PKCS7 * pkcs7, int keyWrapOID, byte * kek, word32 kekSz, byte * keyID, word32 keyIdSz, void * timePtr, byte * otherOID, word32 otherOIDSz, byte * other, word32 otherSz, int options)Adds KEKRI recipient.
int	**wc_PKCS7_SetPassword**(wc_PKCS7 * pkcs7, byte * passwd, word32 pLen)Sets password for PWRI.
int	**wc_PKCS7_AddRecipient_PWRI**(wc_PKCS7 * pkcs7, byte * passwd, word32 pLen, byte * salt, word32 saltSz, int kdfOID, int prfOID, int iterations, int kekEncryptOID, int options)Adds PWRI recipient.
int	**wc_PKCS7_SetOriEncryptCtx**(wc_PKCS7 * pkcs7, void * ctx)Sets originator encryption context.
int	**wc_PKCS7_SetOriDecryptCtx**(wc_PKCS7 * pkcs7, void * ctx)Sets originator decryption context.
int	**wc_PKCS7_SetOriDecryptCb**(wc_PKCS7 * pkcs7, CallbackOriDecrypt cb)Sets originator decryption callback.
int	**wc_PKCS7_AddRecipient_ORI**(wc_PKCS7 * pkcs7, CallbackOriEncrypt cb, int options)Adds ORI recipient.
int	**wc_PKCS7_SetWrapCEKCb**(wc_PKCS7 * pkcs7, CallbackWrapCEK wrapCEKCb)Sets CEK wrap callback.
int	**wc_PKCS7_SetRsaSignRawDigestCb**(wc_PKCS7 * pkcs7, CallbackRsaSignRawDigest cb)Sets RSA sign raw digest callback.

	Name
int	**wc_PKCS7_EncodeAuthEnvelopedData**(wc_PKCS7 * pkcs7, byte * output, word32 outputSz)Encodes authenticated enveloped data.
int	**wc_PKCS7_DecodeAuthEnvelopedData**(wc_PKCS7 * pkcs7, byte * pkiMsg, word32 pkiMsgSz, byte * output, word32 outputSz)Decodes authenticated enveloped data.
int	**wc_PKCS7_EncodeEncryptedData**(wc_PKCS7 * pkcs7, byte * output, word32 outputSz)Encodes encrypted data.
int	**wc_PKCS7_SetDecodeEncryptedCb**(wc_PKCS7 * pkcs7, CallbackDecryptContent decryptionCb)Sets decode encrypted callback.
int	**wc_PKCS7_SetDecodeEncryptedCtx**(wc_PKCS7 * pkcs7, void * ctx)Sets decode encrypted context.
int	**wc_PKCS7_SetStreamMode**(wc_PKCS7 * pkcs7, byte flag, CallbackGetContent getContentCb, CallbackStreamOut streamOutCb, void * ctx)Sets stream mode for PKCS7.
int	**wc_PKCS7_GetStreamMode**(wc_PKCS7 * pkcs7)Gets stream mode setting.
int	**wc_PKCS7_SetNoCerts**(wc_PKCS7 * pkcs7, byte flag)Sets no certificates flag.
int	**wc_PKCS7_GetNoCerts**(wc_PKCS7 * pkcs7)Gets no certificates flag.
int	**wc_PKCS7_EncodeCompressedData**(wc_PKCS7 * pkcs7, byte * output, word32 outputSz)Encodes compressed data.
int	**wc_PKCS7_DecodeCompressedData**(wc_PKCS7 * pkcs7, byte * pkiMsg, word32 pkiMsgSz, byte * output, word32 outputSz)Decodes compressed data.

C.41.3 Types Documentation

C.41.3.1 typedef CallbackAESKeyWrapUnwrap

```
typedef int(* CallbackAESKeyWrapUnwrap) (const byte *key, word32 keySz, const
 ↪  byte *in, word32 inSz, int wrap, byte *out, word32 outSz);
```

Callback used for a custom AES key wrap/unwrap operation.

Parameters:

- **key** Specify the key to use.
- **keySz** Size of the key to use.
- **in** Specify the input data to wrap/unwrap.
- **inSz** Size of the input data.
- **wrap** 1 if the requested operation is a key wrap, 0 for unwrap.
- **out** Specify the output buffer.
- **outSz** Size of the output buffer.

Return: The size of the wrapped/unwrapped key written to the output buffer should be returned on success. A 0 return value or error code (< 0) indicates a failure.

C.41.4 Functions Documentation

C.41.4.1 function wc_PKCS7_InitWithCert

```
int wc_PKCS7_InitWithCert(
    wc_PKCS7 * pkcs7,
    byte * der,
    word32 derSz
)
```

This function initializes a PKCS7 structure with a DER-formatted certificate. To initialize an empty PKCS7 structure, one can pass in a NULL cert and 0 for certSz.

Parameters:

- **pkcs7** pointer to the PKCS7 structure in which to store the decoded cert
- **der** pointer to a buffer containing a DER formatted ASN.1 certificate with which to initialize the PKCS7 structure
- **derSz** size of the certificate buffer

See: wc_PKCS7_Free

Return:

- 0 Returned on successfully initializing the PKCS7 structure
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_PARSE_E Returned if there is an error parsing the cert header
- ASN_OBJECT_ID_E Returned if there is an error parsing the encryption type from the cert
- ASN_EXPECT_0_E Returned if there is a formatting error in the encryption specification of the cert file
- ASN_BEFORE_DATE_E Returned if the date is before the certificate start date
- ASN_AFTER_DATE_E Returned if the date is after the certificate expiration date
- ASN_BITSTR_E Returned if there is an error parsing a bit string from the certificate
- ECC_CURVE_OID_E Returned if there is an error parsing the ECC key from the certificate
- ASN_UNKNOWN_OID_E Returned if the certificate is using an unknown key object id
- ASN_VERSION_E Returned if the ALLOW_V1_EXTENSIONS option is not defined and the certificate is a V1 or V2 certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension
- ASN_CRIT_EXT_E Returned if an unfamiliar critical extension is encountered in processing the certificate
- ASN_SIG_OID_E Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
- ASN_SIG_CONFIRM_E Returned if confirming the certification signature fails
- ASN_NAME_INVALID_E Returned if the certificate's name is not permitted by the CA name constraints
- ASN_NO_SIGNER_E Returned if there is no CA signer to verify the certificate's authenticity

Example

```
wc_PKCS7 pkcs7;
byte derBuff[] = { }; // initialize with DER-encoded certificate
if ( wc_PKCS7_InitWithCert(&pkcs7, derBuff, sizeof(derBuff)) != 0 ) {
    // error parsing certificate into pkcs7 format
}
```

C.41.4.2 function wc_PKCS7_Free

```
void wc_PKCS7_Free(
    wc_PKCS7 * pkcs7
)
```

This function releases any memory allocated by a PKCS7 initializer.

Parameters:

- **pkcs7** pointer to the PKCS7 structure to free

See: wc_PKCS7_InitWithCert

Return: none No returns.

Example

```
PKCS7 pkcs7;
// initialize and use PKCS7 object

wc_PKCS7_Free(pkcs7);
```

C.41.4.3 function wc_PKCS7_EncodeData

```
int wc_PKCS7_EncodeData(
    wc_PKCS7 * pkcs7,
    byte * output,
    word32 outputSz
)
```

This function builds the PKCS7 data content type, encoding the PKCS7 structure into a buffer containing a parsable PKCS7 data packet.

Parameters:

- **pkcs7** pointer to the PKCS7 structure to encode
- **output** pointer to the buffer in which to store the encoded certificate
- **outputSz** size available in the output buffer

See: wc_PKCS7_InitWithCert

Return:

- Success On successfully encoding the PKCS7 data into the buffer, returns the index parsed up to in the PKCS7 structure. This index also corresponds to the bytes written to the output buffer.
- BUFFER_E Returned if the given buffer is not large enough to hold the encoded certificate

Example

```
PKCS7 pkcs7;
int ret;

byte derBuff[] = { }; // initialize with DER-encoded certificate
byte pkcs7Buff[FOURK_BUF];

wc_PKCS7_InitWithCert(&pkcs7, derBuff, sizeof(derBuff));
// update message and data to encode
pkcs7.privateKey = key;
pkcs7.privateKeySz = keySz;
pkcs7.content = data;
pkcs7.contentSz = dataSz;
```

```
... etc.

ret = wc_PKCS7_EncodeData(&pkcs7, pkcs7Buff, sizeof(pkcs7Buff));
if ( ret != 0 ) {
    // error encoding into output buffer
}
```

C.41.4.4 function wc_PKCS7_EncodeSignedData

```
int wc_PKCS7_EncodeSignedData(
    wc_PKCS7 * pkcs7,
    byte * output,
    word32 outputSz
)
```

This function builds the PKCS7 signed data content type, encoding the PKCS7 structure into a buffer containing a parsable PKCS7 signed data packet. For RSA-PSS signers (WC_RSA_PSS), see PKCS#7 RSA-PSS (CMS).

Parameters:

- **pkcs7** pointer to the PKCS7 structure to encode
- **output** pointer to the buffer in which to store the encoded certificate
- **outputSz** size available in the output buffer

See:

- wc_PKCS7_InitWithCert
- wc_PKCS7_VerifySignedData

Return:

- Success On successfully encoding the PKCS7 data into the buffer, returns the index parsed up to in the PKCS7 structure. This index also corresponds to the bytes written to the output buffer.
- BAD_FUNC_ARG Returned if the PKCS7 structure is missing one or more required elements to generate a signed data packet
- MEMORY_E Returned if there is an error allocating memory
- PUBLIC_KEY_E Returned if there is an error parsing the public key
- RSA_BUFFER_E Returned if buffer error, output too small or input too large
- BUFFER_E Returned if the given buffer is not large enough to hold the encoded certificate
- MP_INIT_E may be returned if there is an error generating the signature
- MP_READ_E may be returned if there is an error generating the signature
- MP_CMP_E may be returned if there is an error generating the signature
- MP_INVMOD_E may be returned if there is an error generating the signature
- MP_EXPTMOD_E may be returned if there is an error generating the signature
- MP_MOD_E may be returned if there is an error generating the signature
- MP_MUL_E may be returned if there is an error generating the signature
- MP_ADD_E may be returned if there is an error generating the signature
- MP_MULMOD_E may be returned if there is an error generating the signature
- MP_TO_E may be returned if there is an error generating the signature
- MP_MEM may be returned if there is an error generating the signature

Example

```
PKCS7 pkcs7;
int ret;

byte data[] = {}; // initialize with data to sign
byte derBuff[] = { }; // initialize with DER-encoded certificate
```

```
byte pkcs7Buff[FOURK_BUF];

wc_PKCS7_InitWithCert(&pkcs7, derBuff, sizeof(derBuff));
// update message and data to encode
pkcs7.privateKey = key;
pkcs7.privateKeySz = keySz;
pkcs7.content = data;
pkcs7.contentSz = dataSz;
pkcs7.hashOID = SHAh;
pkcs7.rng = &rng;
... etc.

ret = wc_PKCS7_EncodeSignedData(&pkcs7, pkcs7Buff, sizeof(pkcs7Buff));
if ( ret != 0 ) {
    // error encoding into output buffer
}

wc_PKCS7_Free(&pkcs7);
```

C.41.4.5 function wc_PKCS7_EncodeSignedData_ex

```
int wc_PKCS7_EncodeSignedData_ex(
    wc_PKCS7 * pkcs7,
    const byte * hashBuf,
    word32 hashSz,
    byte * outputHead,
    word32 * outputHeadSz,
    byte * outputFoot,
    word32 * outputFootSz
)
```

This function builds the PKCS7 signed data content type, encoding the PKCS7 structure into a header and footer buffer containing a parsable PKCS7 signed data packet. This does not include the content. A hash must be computed and provided for the data.

Parameters:

- **pkcs7** pointer to the PKCS7 structure to encode
- **hashBuf** pointer to computed hash for the content data
- **hashSz** size of the digest
- **outputHead** pointer to the buffer in which to store the encoded certificate header
- **outputHeadSz** pointer populated with size of output header buffer and returns actual size
- **outputFoot** pointer to the buffer in which to store the encoded certificate footer
- **outputFootSz** pointer populated with size of output footer buffer and returns actual size

See:

- wc_PKCS7_InitWithCert
- wc_PKCS7_VerifySignedData_ex

Return:

- 0=Success
- BAD_FUNC_ARG Returned if the PKCS7 structure is missing one or more required elements to generate a signed data packet
- MEMORY_E Returned if there is an error allocating memory
- PUBLIC_KEY_E Returned if there is an error parsing the public key
- RSA_BUFFER_E Returned if buffer error, output too small or input too large

- BUFFER_E Returned if the given buffer is not large enough to hold the encoded certificate
- MP_INIT_E may be returned if there is an error generating the signature
- MP_READ_E may be returned if there is an error generating the signature
- MP_CMP_E may be returned if there is an error generating the signature
- MP_INVMOD_E may be returned if there is an error generating the signature
- MP_EXPTMOD_E may be returned if there is an error generating the signature
- MP_MOD_E may be returned if there is an error generating the signature
- MP_MUL_E may be returned if there is an error generating the signature
- MP_ADD_E may be returned if there is an error generating the signature
- MP_MULMOD_E may be returned if there is an error generating the signature
- MP_TO_E may be returned if there is an error generating the signature
- MP_MEM may be returned if there is an error generating the signature

Example

```
PKCS7 pkcs7;
int ret;
byte derBuff[] = { }; // initialize with DER-encoded certificate
byte data[] = {}; // initialize with data to sign
byte pkcs7HeadBuff[FOURK_BUF/2];
byte pkcs7FootBuff[FOURK_BUF/2];
word32 pkcs7HeadSz = (word32)sizeof(pkcs7HeadBuff);
word32 pkcs7FootSz = (word32)sizeof(pkcs7HeadBuff);
enum wc_HashType hashType = WC_HASH_TYPE_SHA;
byte   hashBuf[WC_MAX_DIGEST_SIZE];
word32 hashSz = wc_HashGetDigestSize(hashType);

wc_PKCS7_InitWithCert(&pkcs7, derBuff, sizeof(derBuff));
// update message and data to encode
pkcs7.privateKey = key;
pkcs7.privateKeySz = keySz;
pkcs7.content = NULL;
pkcs7.contentSz = dataSz;
pkcs7.hashOID = SHAh;
pkcs7.rng = &rng;
... etc.

// calculate hash for content
ret = wc_HashInit(&hash, hashType);
if (ret == 0) {
    ret = wc_HashUpdate(&hash, hashType, data, sizeof(data));
    if (ret == 0) {
        ret = wc_HashFinal(&hash, hashType, hashBuf);
    }
    wc_HashFree(&hash, hashType);
}

ret = wc_PKCS7_EncodeSignedData_ex(&pkcs7, hashBuf, hashSz, pkcs7HeadBuff,
    &pkcs7HeadSz, pkcs7FootBuff, &pkcs7FootSz);
if ( ret != 0 ) {
    // error encoding into output buffer
}

wc_PKCS7_Free(&pkcs7);
```

C.41.4.6 function wc_PKCS7_VerifySignedData

```
int wc_PKCS7_VerifySignedData(
    wc_PKCS7 * pkcs7,
    byte * pkiMsg,
    word32 pkiMsgSz
)
```

This function takes in a transmitted PKCS7 signed data message, extracts the certificate list and certificate revocation list, and then verifies the signature. It stores the extracted content in the given PKCS7 structure.

Parameters:

- **pkcs7** pointer to the PKCS7 structure in which to store the parsed certificates
- **pkiMsg** pointer to the buffer containing the signed message to verify and decode
- **pkiMsgSz** size of the signed message

See:

- wc_PKCS7_InitWithCert
- wc_PKCS7_EncodeSignedData

Return:

- 0 Returned on successfully extracting the information from the message
- BAD_FUNC_ARG Returned if one of the input parameters is invalid
- ASN_PARSE_E Returned if there is an error parsing the given pkiMsg
- PKCS7_OID_E Returned if the given pkiMsg is not a signed data type
- ASN_VERSION_E Returned if the PKCS7 signer info is not version 1
- MEMORY_E Returned if there is an error allocating memory
- PUBLIC_KEY_E Returned if there is an error parsing the public key
- RSA_BUFFER_E Returned if buffer error, output too small or input too large
- BUFFER_E Returned if the given buffer is not large enough to hold the encoded certificate
- MP_INIT_E may be returned if there is an error generating the signature
- MP_READ_E may be returned if there is an error generating the signature
- MP_CMP_E may be returned if there is an error generating the signature
- MP_INVMOD_E may be returned if there is an error generating the signature
- MP_EXPTMOD_E may be returned if there is an error generating the signature
- MP_MOD_E may be returned if there is an error generating the signature
- MP_MUL_E may be returned if there is an error generating the signature
- MP_ADD_E may be returned if there is an error generating the signature
- MP_MULMOD_E may be returned if there is an error generating the signature
- MP_TO_E may be returned if there is an error generating the signature
- MP_MEM may be returned if there is an error generating the signature

Example

```
PKCS7 pkcs7;
int ret;
byte pkcs7Buff[] = {}; // the PKCS7 signature

wc_PKCS7_InitWithCert(&pkcs7, NULL, 0);
// update message and data to encode
pkcs7.privateKey = key;
pkcs7.privateKeySz = keySz;
pkcs7.content = data;
pkcs7.contentSz = dataSz;
... etc.
```

```
ret = wc_PKCS7_VerifySignedData(&pkcs7, pkcs7Buff, sizeof(pkcs7Buff));
if ( ret != 0 ) {
    // error encoding into output buffer
}

wc_PKCS7_Free(&pkcs7);
```

C.41.4.7 function wc_PKCS7_VerifySignedData_ex

```
int wc_PKCS7_VerifySignedData_ex(
    wc_PKCS7 * pkcs7,
    const byte * hashBuf,
    word32 hashSz,
    byte * pkiMsgHead,
    word32 pkiMsgHeadSz,
    byte * pkiMsgFoot,
    word32 pkiMsgFootSz
)
```

This function takes in a transmitted PKCS7 signed data message as hash/header/footer, then extracts the certificate list and certificate revocation list, and then verifies the signature. It stores the extracted content in the given PKCS7 structure.

Parameters:

- **pkcs7** pointer to the PKCS7 structure in which to store the parsed certificates
- **hashBuf** pointer to computed hash for the content data
- **hashSz** size of the digest
- **pkiMsgHead** pointer to the buffer containing the signed message header to verify and decode
- **pkiMsgHeadSz** size of the signed message header
- **pkiMsgFoot** pointer to the buffer containing the signed message footer to verify and decode
- **pkiMsgFootSz** size of the signed message footer

See:

- wc_PKCS7_InitWithCert
- wc_PKCS7_EncodeSignedData_ex

Return:

- 0 Returned on successfully extracting the information from the message
- BAD_FUNC_ARG Returned if one of the input parameters is invalid
- ASN_PARSE_E Returned if there is an error parsing the given pkiMsg
- PKCS7_OID_E Returned if the given pkiMsg is not a signed data type
- ASN_VERSION_E Returned if the PKCS7 signer info is not version 1
- MEMORY_E Returned if there is an error allocating memory
- PUBLIC_KEY_E Returned if there is an error parsing the public key
- RSA_BUFFER_E Returned if buffer error, output too small or input too large
- BUFFER_E Returned if the given buffer is not large enough to hold the encoded certificate
- MP_INIT_E may be returned if there is an error generating the signature
- MP_READ_E may be returned if there is an error generating the signature
- MP_CMP_E may be returned if there is an error generating the signature
- MP_INVMOD_E may be returned if there is an error generating the signature
- MP_EXPTMOD_E may be returned if there is an error generating the signature
- MP_MOD_E may be returned if there is an error generating the signature
- MP_MUL_E may be returned if there is an error generating the signature
- MP_ADD_E may be returned if there is an error generating the signature

- MP_MULMOD_E may be returned if there is an error generating the signature
- MP_TO_E may be returned if there is an error generating the signature
- MP_MEM may be returned if there is an error generating the signature

Example

```
PKCS7 pkcs7;
int ret;
byte data[] = {}; // initialize with data to sign
byte pkcs7HeadBuff[] = {}; // initialize with PKCS7 header
byte pkcs7FootBuff[] = {}; // initialize with PKCS7 footer
enum wc_HashType hashType = WC_HASH_TYPE_SHA;
byte   hashBuf[WC_MAX_DIGEST_SIZE];
word32 hashSz = wc_HashGetDigestSize(hashType);

wc_PKCS7_InitWithCert(&pkcs7, NULL, 0);
// update message and data to encode
pkcs7.privateKey = key;
pkcs7.privateKeySz = keySz;
pkcs7.content = NULL;
pkcs7.contentSz = dataSz;
pkcs7.rng = &rng;
... etc.

// calculate hash for content
ret = wc_HashInit(&hash, hashType);
if (ret == 0) {
    ret = wc_HashUpdate(&hash, hashType, data, sizeof(data));
    if (ret == 0) {
        ret = wc_HashFinal(&hash, hashType, hashBuf);
    }
    wc_HashFree(&hash, hashType);
}

ret = wc_PKCS7_VerifySignedData_ex(&pkcs7, hashBuf, hashSz, pkcs7HeadBuff,
    sizeof(pkcs7HeadBuff), pkcs7FootBuff, sizeof(pkcs7FootBuff));
if ( ret != 0 ) {
    // error encoding into output buffer
}

wc_PKCS7_Free(&pkcs7);
```

C.41.4.8 function wc_PKCS7_SetAESKeyWrapUnwrapCb

```
int wc_PKCS7_SetAESKeyWrapUnwrapCb(
    wc_PKCS7 * pkcs7,
    CallbackAESKeyWrapUnwrap aesKeyWrapCb
)
```

Set the callback function to be used to perform a custom AES key wrap/unwrap operation.

Parameters:

- **pkcs7** pointer to the PKCS7 structure
- **aesKeyWrapCb** pointer to custom AES key wrap/unwrap function

Returns:

- **0** Callback function was set successfully
- **BAD_FUNC_ARG** Parameter pkcs7 is NULL

C.41.4.9 function wc_PKCS7_EncodeEnvelopedData

```
int wc_PKCS7_EncodeEnvelopedData(
    wc_PKCS7 * pkcs7,
    byte * output,
    word32 outputSz
)
```

This function builds the PKCS7 enveloped data content type, encoding the PKCS7 structure into a buffer containing a parsable PKCS7 enveloped data packet.

Parameters:

- **pkcs7** pointer to the PKCS7 structure to encode
- **output** pointer to the buffer in which to store the encoded certificate
- **outputSz** size available in the output buffer

See:

- wc_PKCS7_InitWithCert
- wc_PKCS7_DecodeEnvelopedData

Return:

- Success Returned on successfully encoding the message in enveloped data format, returns the size written to the output buffer
- BAD_FUNC_ARG: Returned if one of the input parameters is invalid, or if the PKCS7 structure is missing required elements
- ALGO_ID_E Returned if the PKCS7 structure is using an unsupported algorithm type. Currently, only DESb and DES3b are supported
- BUFFER_E Returned if the given output buffer is too small to store the output data
- MEMORY_E Returned if there is an error allocating memory
- RNG_FAILURE_E Returned if there is an error initializing the random number generator for encryption
- DRBG_FAILED Returned if there is an error generating numbers with the random number generator used for encryption
- NOT_COMPILED_IN may be returned if using an ECC key and wolfssl was built without HAVE_X963_KDF support

Example

```
PKCS7 pkcs7;
int ret;

byte derBuff[] = { }; // initialize with DER-encoded certificate
byte pkcs7Buff[FOURK_BUF];

wc_PKCS7_InitWithCert(&pkcs7, derBuff, sizeof(derBuff));
// update message and data to encode
pkcs7.privateKey = key;
pkcs7.privateKeySz = keySz;
pkcs7.content = data;
pkcs7.contentSz = dataSz;
... etc.

ret = wc_PKCS7_EncodeEnvelopedData(&pkcs7, pkcs7Buff, sizeof(pkcs7Buff));
```

```
if ( ret < 0 ) {
    // error encoding into output buffer
}
```

C.41.4.10 function wc_PKCS7_DecodeEnvelopedData

```
int wc_PKCS7_DecodeEnvelopedData(
    wc_PKCS7 * pkcs7,
    byte * pkiMsg,
    word32 pkiMsgSz,
    byte * output,
    word32 outputSz
)
```

This function unwraps and decrypts a PKCS7 enveloped data content type, decoding the message into output. It uses the private key of the PKCS7 object passed in to decrypt the message.

Parameters:

- **pkcs7** pointer to the PKCS7 structure containing the private key with which to decode the enveloped data package
- **pkiMsg** pointer to the buffer containing the enveloped data package
- **pkiMsgSz** size of the enveloped data package
- **output** pointer to the buffer in which to store the decoded message
- **outputSz** size available in the output buffer

See:

- wc_PKCS7_InitWithCert
- wc_PKCS7_EncodeEnvelopedData

Return:

- On successfully extracting the information from the message, returns the bytes written to output
- BAD_FUNC_ARG Returned if one of the input parameters is invalid
- ASN_PARSE_E Returned if there is an error parsing the given pkiMsg
- PKCS7_OID_E Returned if the given pkiMsg is not an enveloped data type
- ASN_VERSION_E Returned if the PKCS7 signer info is not version 0
- MEMORY_E Returned if there is an error allocating memory
- ALGO_ID_E Returned if the PKCS7 structure is using an unsupported algorithm type. Currently, only DESb and DES3b are supported for encryption, with RSAk for signature generation
- PKCS7_RECIP_E Returned if there is no recipient found in the enveloped data that matches the recipient provided
- RSA_BUFFER_E Returned if there is an error during RSA signature verification due to buffer error, output too small or input too large.
- MP_INIT_E may be returned if there is an error during signature verification
- MP_READ_E may be returned if there is an error during signature verification
- MP_CMP_E may be returned if there is an error during signature verification
- MP_INVMOD_E may be returned if there is an error during signature verification
- MP_EXPTMOD_E may be returned if there is an error during signature verification
- MP_MOD_E may be returned if there is an error during signature verification
- MP_MUL_E may be returned if there is an error during signature verification
- MP_ADD_E may be returned if there is an error during signature verification
- MP_MULMOD_E may be returned if there is an error during signature verification
- MP_TO_E may be returned if there is an error during signature verification
- MP_MEM may be returned if there is an error during signature verification
- NOT_COMPILED_IN may be returned if the EnvelopedData is encrypted using an ECC key and wolfssl was built without HAVE_X963_KDF support

Note that if the EnvelopedData is encrypted using an ECC key and the KeyAgreementRecipientInfo structure, then either the HAVE_AES_KEYWRAP build option should be enabled to enable the wolfcrypt built-in AES key wrap/unwrap functionality, or a custom AES key wrap/unwrap callback should be set with wc_PKCS7_SetAESKeyWrapUnwrapCb(). If neither of these is true, decryption will fail.

Example

```
PKCS7 pkcs7;
byte received[] = { }; // initialize with received enveloped message
byte decoded[FOURK_BUF];
int decodedSz;

// initialize pkcs7 with certificate
// update key
pkcs7.privateKey = key;
pkcs7.privateKeySz = keySz;

decodedSz = wc_PKCS7_DecodeEnvelopedData(&pkcs7, received,
    sizeof(received),decoded, sizeof(decoded));
if ( decodedSz < 0 ) {
    // error decoding message
}
```

C.41.4.11 function wc_PKCS7_GetEnvelopedDataKariRid

```
int wc_PKCS7_GetEnvelopedDataKariRid(
    const byte * in,
    word32 inSz,
    byte * out,
    word32 * outSz
)
```

This function extracts the KeyAgreeRecipientIdentifier object from an EnvelopedData package containing a KeyAgreeRecipientInfo RecipientInfo object. Only the first KeyAgreeRecipientIdentifer found in the first RecipientInfo is copied. This function does not support multiple RecipientInfo objects or multiple RecipientEncryptedKey objects within an KeyAgreeRecipientInfo.

Parameters:

- **in** Input buffer containing the EnvelopedData ContentInfo message.
- **inSz** Size of the input buffer.
- **out** Output buffer.
- **outSz** Output buffer size on input, Size written on output.

Return:

- Returns 0 on success.
- BAD_FUNC_ARG Returned if one of the input parameters is invalid.
- ASN_PARSE_E Returned if there is an error parsing the input message.
- PKCS7_OID_E Returned if the input message is not an enveloped data type.
- BUFFER_E Returned if there is not enough room in the output buffer.

C.41.4.12 function wc_PKCS7_DecodeEncryptedData

```
int wc_PKCS7_DecodeEncryptedData(
    wc_PKCS7 * pkcs7,
    byte * pkiMsg,
    word32 pkiMsgSz,
```

```
    byte * output,
    word32 outputSz
)
```

This function unwraps and decrypts a PKCS7 encrypted data content type, decoding the message into output. It uses the encryption key of the PKCS7 object passed in via pkcs7->encryptionKey and pkcs7->encryptionKeySz to decrypt the message.

Parameters:

- **pkcs7** pointer to the PKCS7 structure containing the encryption key with which to decode the encrypted data package
- **pkiMsg** pointer to the buffer containing the encrypted data package
- **pkiMsgSz** size of the encrypted data package
- **output** pointer to the buffer in which to store the decoded message
- **outputSz** size available in the output buffer

See: wc_PKCS7_InitWithCert

Return:

- On successfully extracting the information from the message, returns the bytes written to output
- BAD_FUNC_ARG Returned if one of the input parameters is invalid
- ASN_PARSE_E Returned if there is an error parsing the given pkiMsg
- PKCS7_OID_E Returned if the given pkiMsg is not an encrypted data type
- ASN_VERSION_E Returned if the PKCS7 signer info is not version 0
- MEMORY_E Returned if there is an error allocating memory
- BUFFER_E Returned if the encrypted content size is invalid

Example

```
PKCS7 pkcs7;
byte received[] = { }; // initialize with received encrypted data message
byte decoded[FOURK_BUF];
int decodedSz;

// initialize pkcs7 with certificate
// update key
pkcs7.encryptionKey = key;
pkcs7.encryptionKeySz = keySz;

decodedSz = wc_PKCS7_DecodeEncryptedData(&pkcs7, received,
    sizeof(received), decoded, sizeof(decoded));
if ( decodedSz < 0 ) {
    // error decoding message
}
```

C.41.4.13 function wc_PKCS7_DecodeEncryptedKeyPackage

```
int wc_PKCS7_DecodeEncryptedKeyPackage(
    wc_PKCS7 * pkcs7,
    byte * pkiMsg,
    word32 pkiMsgSz,
    byte * output,
    word32 outputSz
)
```

This function unwraps and decrypts a PKCS7 encrypted key package content type, decoding the message into output. If the wrapped content type is EncryptedData, the encryption key must be set in the

pkcs7 input structure (via pkcs7->encryptionKey and pkcs7->encryptionKeySz). If the wrapped content type is EnvelopedData, the private key must be set in the pkcs7 input structure (via pkcs7->privateKey and pkcs7->privateKeySz). A wrapped content type of AuthEnvelopedData is not currently supported.

Parameters:

- **pkcs7** pointer to the PKCS7 structure containing the private key or encryption key with which to decode the encrypted key package
- **pkiMsg** pointer to the buffer containing the encrypted key package message
- **pkiMsgSz** size of the encrypted key package message
- **output** pointer to the buffer in which to store the decoded output
- **outputSz** size available in the output buffer

See: wc_PKCS7_InitWithCert

Return:

- On successfully extracting the information from the message, returns the bytes written to output
- BAD_FUNC_ARG Returned if one of the input parameters is invalid
- ASN_PARSE_E Returned if there is an error parsing the given pkiMsg or if the wrapped content type is EncryptedData and support for EncryptedData is not compiled in (e.g. NO_PKCS7_ENCRYPTED_DATA is set)
- PKCS7_OID_E Returned if the given pkiMsg is not an encrypted key package data type

This function will automatically call either wc_PKCS7_DecodeEnvelopedData() or wc_PKCS7_DecodeEncryptedData() depending on the wrapped content type. This function could also return any error code from either of those functions in addition to the error codes listed here.

Example

```
PKCS7 pkcs7;
byte received[] = { }; // initialize with received encrypted data message
byte decoded[FOURK_BUF];
int decodedSz;

// initialize pkcs7 with certificate
// update key for expected EnvelopedData (example)
pkcs7.privateKey = key;
pkcs7.privateKeySz = keySz;

decodedSz = wc_PKCS7_DecodeEncryptedKeyPackage(&pkcs7, received,
    sizeof(received), decoded, sizeof(decoded));
if ( decodedSz < 0 ) {
    // error decoding message
}
```

C.41.4.14 function wc_PKCS7_DecodeSymmetricKeyPackageAttribute

```
int wc_PKCS7_DecodeSymmetricKeyPackageAttribute(
    const byte * skp,
    word32 skpSz,
    size_t index,
    const byte ** attr,
    word32 * attrSz
)
```

This function provides access to a SymmetricKeyPackage attribute.

Parameters:

- **skp** Input buffer containing the SymmetricKeyPackage object.
- **skpSz** Size of the SymmetricKeyPackage object.
- **index** Index of the attribute to access.
- **attr** Buffer in which to store the pointer to the requested attribute object.
- **attrSz** Buffer in which to store the size of the requested attribute object.

Return:

- 0 The requested attribute has been successfully located. attr and attrSz output variables are populated with the address and size of the attribute. The attribute will be in the same buffer passed in via the skp input pointer.
- BAD_FUNC_ARG One of the input parameters is invalid.
- ASN_PARSE_E An error was encountered parsing the input object.
- BAD_INDEX_E The requested attribute index was invalid.

C.41.4.15 function wc_PKCS7_DecodeSymmetricKeyPackageKey

```
int wc_PKCS7_DecodeSymmetricKeyPackageKey(
    const byte * skp,
    word32 skpSz,
    size_t index,
    const byte ** key,
    word32 * keySz
)
```

This function provides access to a SymmetricKeyPackage key.

Parameters:

- **skp** Input buffer containing the SymmetricKeyPackage object.
- **skpSz** Size of the SymmetricKeyPackage object.
- **index** Index of the key to access.
- **key** Buffer in which to store the pointer to the requested key object.
- **keySz** Buffer in which to store the size of the requested key object.

Return:

- 0 The requested key has been successfully located. key and keySz output variables are populated with the address and size of the key. The key will be in the same buffer passed in via the skp input pointer.
- BAD_FUNC_ARG One of the input parameters is invalid.
- ASN_PARSE_E An error was encountered parsing the input object.
- BAD_INDEX_E The requested key index was invalid.

C.41.4.16 function wc_PKCS7_DecodeOneSymmetricKeyAttribute

```
int wc_PKCS7_DecodeOneSymmetricKeyAttribute(
    const byte * osk,
    word32 oskSz,
    size_t index,
    const byte ** attr,
    word32 * attrSz
)
```

This function provides access to a OneSymmetricKey attribute.

Parameters:

- **osk** Input buffer containing the OneSymmetricKey object.

- **oskSz** Size of the OneSymmetricKey object.
- **index** Index of the attribute to access.
- **attr** Buffer in which to store the pointer to the requested attribute object.
- **attrSz** Buffer in which to store the size of the requested attribute object.

Return:

- 0 The requested attribute has been successfully located. attr and attrSz output variables are populated with the address and size of the attribute. The attribute will be in the same buffer passed in via the osk input pointer.
- BAD_FUNC_ARG One of the input parameters is invalid.
- ASN_PARSE_E An error was encountered parsing the input object.
- BAD_INDEX_E The requested attribute index was invalid.

C.41.4.17 function wc_PKCS7_DecodeOneSymmetricKeyKey

```
int wc_PKCS7_DecodeOneSymmetricKeyKey(
    const byte * osk,
    word32 oskSz,
    const byte ** key,
    word32 * keySz
)
```

This function provides access to a OneSymmetricKey key.

Parameters:

- **osk** Input buffer containing the OneSymmetricKey object.
- **oskSz** Size of the OneSymmetricKey object.
- **key** Buffer in which to store the pointer to the requested key object.
- **keySz** Buffer in which to store the size of the requested key object.

Return:

- 0 The requested key has been successfully located. key and keySz output variables are populated with the address and size of the key. The key will be in the same buffer passed in via the osk input pointer.
- BAD_FUNC_ARG One of the input parameters is invalid.
- ASN_PARSE_E An error was encountered parsing the input object.

C.41.4.18 function wolfSSL_PKCS7_new

```
PKCS7 * wolfSSL_PKCS7_new(
    void
)
```

Creates new PKCS7 structure.

Parameters:

- **none** No parameters

See: wolfSSL_PKCS7_free

Return:

- Pointer to new PKCS7 structure on success
- NULL on error

Example

```
PKCS7* pkcs7 = wolfSSL_PKCS7_new();
if (pkcs7 != NULL) {
    // use pkcs7
    wolfSSL_PKCS7_free(pkcs7);
}
```

C.41.4.19 function wolfSSL_PKCS7_SIGNED_new

```
PKCS7_SIGNED * wolfSSL_PKCS7_SIGNED_new(
    void
)
```

Creates new PKCS7_SIGNED structure.

Parameters:

- **none** No parameters

See: wolfSSL_PKCS7_SIGNED_free

Return:

- Pointer to new PKCS7_SIGNED structure on success
- NULL on error

Example

```
PKCS7_SIGNED* p7 = wolfSSL_PKCS7_SIGNED_new();
if (p7 != NULL) {
    // use p7
    wolfSSL_PKCS7_SIGNED_free(p7);
}
```

C.41.4.20 function wolfSSL_PKCS7_free

```
void wolfSSL_PKCS7_free(
    PKCS7 * p7
)
```

Frees PKCS7 structure.

Parameters:

- **p7** PKCS7 structure to free

See: wolfSSL_PKCS7_new

Return: none No returns

Example

```
PKCS7* pkcs7 = wolfSSL_PKCS7_new();
wolfSSL_PKCS7_free(pkcs7);
```

C.41.4.21 function wolfSSL_PKCS7_SIGNED_free

```
void wolfSSL_PKCS7_SIGNED_free(
    PKCS7_SIGNED * p7
)
```

Frees PKCS7_SIGNED structure.

Parameters:

- **p7** PKCS7_SIGNED structure to free

See: wolfSSL_PKCS7_SIGNED_new

Return: none No returns

Example

```
PKCS7_SIGNED* p7 = wolfSSL_PKCS7_SIGNED_new();
wolfSSL_PKCS7_SIGNED_free(p7);
```

C.41.4.22 function wolfSSL_d2i_PKCS7

```
PKCS7 * wolfSSL_d2i_PKCS7(
    PKCS7 ** p7,
    const unsigned char ** in,
    int len
)
```

Decodes DER-encoded PKCS7 structure.

Parameters:

- **p7** Pointer to PKCS7 pointer (can be NULL)
- **in** Pointer to DER-encoded data
- **len** Length of DER data

See: wolfSSL_i2d_PKCS7

Return:

- Pointer to decoded PKCS7 structure on success
- NULL on error

Example

```
PKCS7* p7 = NULL;
const unsigned char* der = ...; // DER data
p7 = wolfSSL_d2i_PKCS7(&p7, &der, derLen);
```

C.41.4.23 function wolfSSL_d2i_PKCS7_bio

```
PKCS7 * wolfSSL_d2i_PKCS7_bio(
    WOLFSSL_BIO * bio,
    PKCS7 ** p7
)
```

Decodes PKCS7 from BIO.

Parameters:

- **bio** BIO to read from
- **p7** Pointer to PKCS7 pointer (can be NULL)

See: wolfSSL_i2d_PKCS7_bio

Return:

- Pointer to decoded PKCS7 structure on success
- NULL on error

Example

```
WOLFSSL_BIO* bio = wolfSSL_BIO_new_file("pkcs7.der", "rb");
PKCS7* p7 = wolfSSL_d2i_PKCS7_bio(bio, NULL);
```

C.41.4.24 function wolfSSL_i2d_PKCS7_bio

```
int wolfSSL_i2d_PKCS7_bio(
    WOLFSSL_BIO * bio,
    PKCS7 * p7
)
```

Encodes PKCS7 to BIO.

Parameters:

- **bio** BIO to write to
- **p7** PKCS7 structure to encode

See: wolfSSL_d2i_PKCS7_bio

Return:

- Length written on success
- negative on error

Example

```
WOLFSSL_BIO* bio = wolfSSL_BIO_new(wolfSSL_BIO_s_mem());
int ret = wolfSSL_i2d_PKCS7_bio(bio, p7);
```

C.41.4.25 function wolfSSL_i2d_PKCS7

```
int wolfSSL_i2d_PKCS7(
    PKCS7 * p7,
    unsigned char ** out
)
```

Encodes PKCS7 to DER.

Parameters:

- **p7** PKCS7 structure to encode
- **out** Pointer to output buffer pointer

See: wolfSSL_d2i_PKCS7

Return:

- Length written on success
- negative on error

Example

```
unsigned char* der = NULL;
int len = wolfSSL_i2d_PKCS7(p7, &der);
```

C.41.4.26 function wolfSSL_PKCS7_sign

```
PKCS7 * wolfSSL_PKCS7_sign(
    WOLFSSL_X509 * signer,
    WOLFSSL_EVP_PKEY * pkey,
    WOLFSSL_STACK * certs,
    WOLFSSL_BIO * in,
```

```
    int flags
)
```

Creates signed PKCS7 message.

Parameters:

- **signer** Signer certificate
- **pkey** Private key
- **certs** Additional certificates
- **in** Input data BIO
- **flags** Operation flags

See: wolfSSL_PKCS7_verify

Return:

- Pointer to signed PKCS7 structure on success
- NULL on error

Example

```
PKCS7* p7 = wolfSSL_PKCS7_sign(cert, pkey, NULL, bio, 0);
```

C.41.4.27 function wolfSSL_PKCS7_verify

```
int wolfSSL_PKCS7_verify(
    PKCS7 * p7,
    WOLFSSL_STACK * certs,
    WOLFSSL_X509_STORE * store,
    WOLFSSL_BIO * in,
    WOLFSSL_BIO * out,
    int flags
)
```

Verifies signed PKCS7 message.

Parameters:

- **p7** PKCS7 structure to verify
- **certs** Certificate stack
- **store** Certificate store
- **in** Input data BIO
- **out** Output BIO
- **flags** Operation flags

See: wolfSSL_PKCS7_sign

Return:

- 1 on success
- 0 or negative on error

Example

```
int ret = wolfSSL_PKCS7_verify(p7, NULL, store, NULL, out, 0);
```

C.41.4.28 function wolfSSL_PKCS7_final

```
int wolfSSL_PKCS7_final(
    PKCS7 * pkcs7,
    WOLFSSL_BIO * in,
```

```
    int flags
)
```

Finalizes PKCS7 structure with data.

Parameters:

- **pkcs7** PKCS7 structure
- **in** Input data BIO
- **flags** Operation flags

See: wolfSSL_PKCS7_sign

Return:

- 1 on success
- 0 or negative on error

Example

```
int ret = wolfSSL_PKCS7_final(pkcs7, bio, 0);
```

C.41.4.29 function wolfSSL_PKCS7_encode_certs

```
int wolfSSL_PKCS7_encode_certs(
    PKCS7 * p7,
    WOLFSSL_STACK * certs,
    WOLFSSL_BIO * out
)
```

Encodes certificates into PKCS7.

Parameters:

- **p7** PKCS7 structure
- **certs** Certificate stack
- **out** Output BIO

See: wolfSSL_PKCS7_to_stack

Return:

- 1 on success
- 0 or negative on error

Example

```
int ret = wolfSSL_PKCS7_encode_certs(p7, certs, bio);
```

C.41.4.30 function wolfSSL_PKCS7_to_stack

```
WOLFSSL_STACK * wolfSSL_PKCS7_to_stack(
    PKCS7 * pkcs7
)
```

Converts PKCS7 certificates to stack.

Parameters:

- **pkcs7** PKCS7 structure

See: wolfSSL_PKCS7_encode_certs

Return:

- Pointer to certificate stack on success
- NULL on error

Example

```
WOLFSSL_STACK* certs = wolfSSL_PKCS7_to_stack(pkcs7);
```

C.41.4.31 function wolfSSL_PKCS7_get0_signers

```
WOLFSSL_STACK * wolfSSL_PKCS7_get0_signers(
    PKCS7 * p7,
    WOLFSSL_STACK * certs,
    int flags
)
```

Gets signer certificates from PKCS7.

Parameters:

- **p7** PKCS7 structure
- **certs** Certificate stack
- **flags** Operation flags

See: wolfSSL_PKCS7_verify

Return:

- Pointer to signer certificate stack on success
- NULL on error

Example

```
WOLFSSL_STACK* signers = wolfSSL_PKCS7_get0_signers(p7, NULL, 0);
```

C.41.4.32 function wolfSSL_PEM_write_bio_PKCS7

```
int wolfSSL_PEM_write_bio_PKCS7(
    WOLFSSL_BIO * bio,
    PKCS7 * p7
)
```

Writes PKCS7 to BIO in PEM format.

Parameters:

- **bio** Output BIO
- **p7** PKCS7 structure

See: wolfSSL_SMIME_write_PKCS7

Return:

- 1 on success
- 0 or negative on error

Example

```
int ret = wolfSSL_PEM_write_bio_PKCS7(bio, p7);
```

C.41.4.33 function wolfSSL_SMIME_read_PKCS7

```
PKCS7 * wolfSSL_SMIME_read_PKCS7(
    WOLFSSL_BIO * in,
    WOLFSSL_BIO ** bcont
)
```

Reads S/MIME PKCS7 from BIO.

Parameters:

- **in** Input BIO
- **bcont** Pointer to content BIO pointer

See: wolfSSL_SMIME_write_PKCS7

Return:

- Pointer to PKCS7 structure on success
- NULL on error

Example

```
WOLFSSL_BIO* cont = NULL;
PKCS7* p7 = wolfSSL_SMIME_read_PKCS7(bio, &cont);
```

C.41.4.34 function wolfSSL_SMIME_write_PKCS7

```
int wolfSSL_SMIME_write_PKCS7(
    WOLFSSL_BIO * out,
    PKCS7 * pkcs7,
    WOLFSSL_BIO * in,
    int flags
)
```

Writes PKCS7 to BIO in S/MIME format.

Parameters:

- **out** Output BIO
- **pkcs7** PKCS7 structure
- **in** Input data BIO
- **flags** Operation flags

See: wolfSSL_SMIME_read_PKCS7

Return:

- 1 on success
- 0 or negative on error

Example

```
int ret = wolfSSL_SMIME_write_PKCS7(out, pkcs7, in, 0);
```

C.41.4.35 function wc_PKCS7_New

```
wc_PKCS7 * wc_PKCS7_New(
    void * heap,
    int devId
)
```

Creates new wc_PKCS7 structure.

Parameters:

- **heap** Heap hint
- **devId** Device ID

See: wc_PKCS7_Init

Return:

- Pointer to new wc_PKCS7 structure on success
- NULL on error

Example

```
wc_PKCS7* pkcs7 = wc_PKCS7_New(NULL, INVALID_DEVID);
```

C.41.4.36 function wc_PKCS7_SetUnknownExtCallback

```
void wc_PKCS7_SetUnknownExtCallback(
    wc_PKCS7 * pkcs7,
    wc_UnknownExtCallback cb
)
```

Sets unknown extension callback.

Parameters:

- **pkcs7** PKCS7 structure
- **cb** Callback function

See: wc_PKCS7_Init

Return: none No returns

Example

```
wc_PKCS7_SetUnknownExtCallback(pkcs7, myCallback);
```

C.41.4.37 function wc_PKCS7_Init

```
int wc_PKCS7_Init(
    wc_PKCS7 * pkcs7,
    void * heap,
    int devId
)
```

Initializes wc_PKCS7 structure.

Parameters:

- **pkcs7** PKCS7 structure
- **heap** Heap hint
- **devId** Device ID

See: wc_PKCS7_New

Return:

- 0 on success
- negative on error

Example

```
wc_PKCS7 pkcs7;
int ret = wc_PKCS7_Init(&pkcs7, NULL, INVALID_DEVID);
```

C.41.4.38 function wc_PKCS7_AddCertificate

```
int wc_PKCS7_AddCertificate(
    wc_PKCS7 * pkcs7,
    byte * der,
    word32 derSz
)
```

Adds certificate to PKCS7.

Parameters:

- **pkcs7** PKCS7 structure
- **der** DER-encoded certificate
- **derSz** Certificate size

See: wc_PKCS7_Init

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_AddCertificate(&pkcs7, cert, certSz);
```

C.41.4.39 function wc_PKCS7_GetAttributeValue

```
int wc_PKCS7_GetAttributeValue(
    wc_PKCS7 * pkcs7,
    const byte * oid,
    word32 oidSz,
    byte * out,
    word32 * outSz
)
```

Gets attribute value from PKCS7.

Parameters:

- **pkcs7** PKCS7 structure
- **oid** Attribute OID
- **oidSz** OID size
- **out** Output buffer
- **outSz** Output buffer size pointer

See: wc_PKCS7_Init

Return:

- 0 on success
- negative on error

Example

```
byte value[256];
word32 valueSz = sizeof(value);
```

```
int ret = wc_PKCS7_GetAttributeValue(&pkcs7, oid, oidSz, value,
                                     &valueSz);
```

C.41.4.40 function wc_PKCS7_SetSignerIdentifierType

```
int wc_PKCS7_SetSignerIdentifierType(
    wc_PKCS7 * pkcs7,
    int type
)
```

Sets signer identifier type.

Parameters:

- **pkcs7** PKCS7 structure
- **type** Identifier type

See: wc_PKCS7_Init

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_SetSignerIdentifierType(&pkcs7, CMS_SKID);
```

C.41.4.41 function wc_PKCS7_SetContentType

```
int wc_PKCS7_SetContentType(
    wc_PKCS7 * pkcs7,
    byte * contentType,
    word32 sz
)
```

Sets content type.

Parameters:

- **pkcs7** PKCS7 structure
- **contentType** Content type OID
- **sz** OID size

See: wc_PKCS7_Init

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_SetContentType(&pkcs7, DATA, sizeof(DATA));
```

C.41.4.42 function wc_PKCS7_GetPadSize

```
int wc_PKCS7_GetPadSize(
    word32 inputSz,
    word32 blockSz
)
```

Gets padding size for block cipher.

Parameters:

- **inputSz** Input size
- **blockSz** Block size

See: wc_PKCS7_PadData

Return: Padding size

Example

```
int padSz = wc_PKCS7_GetPadSize(dataSz, AES_BLOCK_SIZE);
```

C.41.4.43 function wc_PKCS7_PadData

```
int wc_PKCS7_PadData(
    byte * in,
    word32 inSz,
    byte * out,
    word32 outSz,
    word32 blockSz
)
```

Pads data for block cipher.

Parameters:

- **in** Input data
- **inSz** Input size
- **out** Output buffer
- **outSz** Output buffer size
- **blockSz** Block size

See: wc_PKCS7_GetPadSize

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_PadData(data, dataSz, padded, paddedSz,
                           AES_BLOCK_SIZE);
```

C.41.4.44 function wc_PKCS7_SetCustomSKID

```
int wc_PKCS7_SetCustomSKID(
    wc_PKCS7 * pkcs7,
    const byte * in,
    word16 inSz
)
```

Sets custom subject key identifier.

Parameters:

- **pkcs7** PKCS7 structure
- **in** SKID data
- **inSz** SKID size

See: wc_PKCS7_Init

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_SetCustomSKID(&pkcs7, skid, skidSz);
```

C.41.4.45 function wc_PKCS7_SetDetached

```
int wc_PKCS7_SetDetached(
    wc_PKCS7 * pkcs7,
    word16 flag
)
```

Sets detached signature flag.

Parameters:

- **pkcs7** PKCS7 structure
- **flag** Detached flag (1=detached, 0=attached)

See: wc_PKCS7_Init

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_SetDetached(&pkcs7, 1);
```

C.41.4.46 function wc_PKCS7_NoDefaultSignedAttribs

```
int wc_PKCS7_NoDefaultSignedAttribs(
    wc_PKCS7 * pkcs7
)
```

Disables default signed attributes.

Parameters:

- **pkcs7** PKCS7 structure

See: wc_PKCS7_SetDefaultSignedAttribs

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_NoDefaultSignedAttribs(&pkcs7);
```

C.41.4.47 function wc_PKCS7_SetDefaultSignedAttribs

```
int wc_PKCS7_SetDefaultSignedAttribs(
    wc_PKCS7 * pkcs7,
    word16 flag
)
```

Sets default signed attributes flag.

Parameters:

- **pkcs7** PKCS7 structure
- **flag** Default attributes flag

See: wc_PKCS7_NoDefaultSignedAttribs

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_SetDefaultSignedAttribs(&pkcs7, 1);
```

C.41.4.48 function wc_PKCS7_AllowDegenerate

```
void wc_PKCS7_AllowDegenerate(
    wc_PKCS7 * pkcs7,
    word16 flag
)
```

Allows degenerate PKCS7 (no signers).

Parameters:

- **pkcs7** PKCS7 structure
- **flag** Allow degenerate flag

See: wc_PKCS7_Init

Return: none No returns

Example

```
wc_PKCS7_AllowDegenerate(&pkcs7, 1);
```

C.41.4.49 function wc_PKCS7_GetSignerSID

```
int wc_PKCS7_GetSignerSID(
    wc_PKCS7 * pkcs7,
    byte * out,
    word32 * outSz
)
```

Gets signer subject identifier.

Parameters:

- **pkcs7** PKCS7 structure
- **out** Output buffer
- **outSz** Output buffer size pointer

See: wc_PKCS7_Init

Return:

- 0 on success
- negative on error

Example

```
byte sid[256];
word32 sidSz = sizeof(sid);
int ret = wc_PKCS7_GetSignerSID(&pkcs7, sid, &sidSz);
```

C.41.4.50 function wc_PKCS7_EncodeSignedFPD

```
int wc_PKCS7_EncodeSignedFPD(
    wc_PKCS7 * pkcs7,
    byte * privateKey,
    word32 privateKeySz,
    int signOID,
    int hashOID,
    byte * content,
    word32 contentSz,
    PKCS7Attrib * signedAttribs,
    word32 signedAttribsSz,
    byte * output,
    word32 outputSz
)
```

Encodes signed FirmwarePackageData.

Parameters:

- **pkcs7** PKCS7 structure
- **privateKey** Private key
- **privateKeySz** Private key size
- **signOID** Signature algorithm OID
- **hashOID** Hash algorithm OID
- **content** Content data
- **contentSz** Content size
- **signedAttribs** Signed attributes
- **signedAttribsSz** Signed attributes count
- **output** Output buffer
- **outputSz** Output buffer size

See: wc_PKCS7_EncodeSignedData

Return:

- Size of encoded data on success
- negative on error

Example

```
int ret = wc_PKCS7_EncodeSignedFPD(&pkcs7, key, keySz, RSAk, SHAh,
                                   data, dataSz, NULL, 0, out, outSz);
```

C.41.4.51 function wc_PKCS7_EncodeSignedEncryptedFPD

```
int wc_PKCS7_EncodeSignedEncryptedFPD(
    wc_PKCS7 * pkcs7,
    byte * encryptKey,
    word32 encryptKeySz,
    byte * privateKey,
    word32 privateKeySz,
    int encryptOID,
    int signOID,
    int hashOID,
    byte * content,
    word32 contentSz,
    PKCS7Attrib * unprotectedAttribs,
    word32 unprotectedAttribsSz,
    PKCS7Attrib * signedAttribs,
    word32 signedAttribsSz,
    byte * output,
    word32 outputSz
)
```

Encodes signed encrypted FirmwarePackageData.

Parameters:

- **pkcs7** PKCS7 structure
- **encryptKey** Encryption key
- **encryptKeySz** Encryption key size
- **privateKey** Private key
- **privateKeySz** Private key size
- **encryptOID** Encryption algorithm OID
- **signOID** Signature algorithm OID
- **hashOID** Hash algorithm OID
- **content** Content data
- **contentSz** Content size
- **unprotectedAttribs** Unprotected attributes
- **unprotectedAttribsSz** Unprotected attributes count
- **signedAttribs** Signed attributes
- **signedAttribsSz** Signed attributes count
- **output** Output buffer
- **outputSz** Output buffer size

See: wc_PKCS7_EncodeSignedFPD

Return:

- Size of encoded data on success
- negative on error

Example

```
int ret = wc_PKCS7_EncodeSignedEncryptedFPD(&pkcs7, encKey, encKeySz,
                                            key, keySz, AES256CBCb,
                                            RSAk, SHAh, data, dataSz,
                                            NULL, 0, NULL, 0, out,
                                            outSz);
```

C.41.4.52 function wc_PKCS7_EncodeSignedCompressedFPD

```
int wc_PKCS7_EncodeSignedCompressedFPD(
    wc_PKCS7 * pkcs7,
    byte * privateKey,
    word32 privateKeySz,
    int signOID,
    int hashOID,
    byte * content,
    word32 contentSz,
    PKCS7Attrib * signedAttribs,
    word32 signedAttribsSz,
    byte * output,
    word32 outputSz
)
```

Encodes signed compressed FirmwarePackageData.

Parameters:

- **pkcs7** PKCS7 structure
- **privateKey** Private key
- **privateKeySz** Private key size
- **signOID** Signature algorithm OID
- **hashOID** Hash algorithm OID
- **content** Content data
- **contentSz** Content size
- **signedAttribs** Signed attributes
- **signedAttribsSz** Signed attributes count
- **output** Output buffer
- **outputSz** Output buffer size

See: wc_PKCS7_EncodeSignedFPD

Return:

- Size of encoded data on success
- negative on error

Example

```
int ret = wc_PKCS7_EncodeSignedCompressedFPD(&pkcs7, key, keySz, RSAk,
                                             SHAh, data, dataSz, NULL,
                                             0, out, outSz);
```

C.41.4.53 function wc_PKCS7_EncodeSignedEncryptedCompressedFPD

```
int wc_PKCS7_EncodeSignedEncryptedCompressedFPD(
    wc_PKCS7 * pkcs7,
    byte * encryptKey,
    word32 encryptKeySz,
    byte * privateKey,
    word32 privateKeySz,
    int encryptOID,
    int signOID,
    int hashOID,
    byte * content,
    word32 contentSz,
    PKCS7Attrib * unprotectedAttribs,
    word32 unprotectedAttribsSz,
```

```
    PKCS7Attrib * signedAttribs,
    word32 signedAttribsSz,
    byte * output,
    word32 outputSz
)
```

Encodes signed encrypted compressed FirmwarePackageData.

Parameters:

- **pkcs7** PKCS7 structure
- **encryptKey** Encryption key
- **encryptKeySz** Encryption key size
- **privateKey** Private key
- **privateKeySz** Private key size
- **encryptOID** Encryption algorithm OID
- **signOID** Signature algorithm OID
- **hashOID** Hash algorithm OID
- **content** Content data
- **contentSz** Content size
- **unprotectedAttribs** Unprotected attributes
- **unprotectedAttribsSz** Unprotected attributes count
- **signedAttribs** Signed attributes
- **signedAttribsSz** Signed attributes count
- **output** Output buffer
- **outputSz** Output buffer size

See: wc_PKCS7_EncodeSignedCompressedFPD

Return:

- Size of encoded data on success
- negative on error

Example

```
int ret = wc_PKCS7_EncodeSignedEncryptedCompressedFPD(&pkcs7, encKey,
                                                      encKeySz, key,
                                                      keySz, AES256CBCb,
                                                      RSAk, SHAh, data,
                                                      dataSz, NULL, 0,
                                                      NULL, 0, out,
                                                      outSz);
```

C.41.4.54 function wc_PKCS7_AddRecipient_KTRI

```
int wc_PKCS7_AddRecipient_KTRI(
    wc_PKCS7 * pkcs7,
    const byte * cert,
    word32 certSz,
    int options
)
```

Adds KTRI recipient.

Parameters:

- **pkcs7** PKCS7 structure
- **cert** Recipient certificate
- **certSz** Certificate size

- **options** Options flags

See: wc_PKCS7_AddRecipient_KARI

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_AddRecipient_KTRI(&pkcs7, cert, certSz, 0);
```

C.41.4.55 function wc_PKCS7_AddRecipient_KARI

```
int wc_PKCS7_AddRecipient_KARI(
    wc_PKCS7 * pkcs7,
    const byte * cert,
    word32 certSz,
    int keyWrapOID,
    int keyAgreeOID,
    byte * ukm,
    word32 ukmSz,
    int options
)
```

Adds KARI recipient.

Parameters:

- **pkcs7** PKCS7 structure
- **cert** Recipient certificate
- **certSz** Certificate size
- **keyWrapOID** Key wrap algorithm OID
- **keyAgreeOID** Key agreement algorithm OID
- **ukm** User keying material
- **ukmSz** UKM size
- **options** Options flags

See: wc_PKCS7_AddRecipient_KTRI

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_AddRecipient_KARI(&pkcs7, cert, certSz, AES256_WRAP,
                                     dhSinglePass_stdDH_sha256kdf_scheme,
                                     NULL, 0, 0);
```

C.41.4.56 function wc_PKCS7_SetKey

```
int wc_PKCS7_SetKey(
    wc_PKCS7 * pkcs7,
    byte * key,
    word32 keySz
)
```

Sets encryption key.

Parameters:

- **pkcs7** PKCS7 structure
- **key** Encryption key
- **keySz** Key size

See: wc_PKCS7_Init

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_SetKey(&pkcs7, key, keySz);
```

C.41.4.57 function wc_PKCS7_AddRecipient_KEKRI

```
int wc_PKCS7_AddRecipient_KEKRI(
    wc_PKCS7 * pkcs7,
    int keyWrapOID,
    byte * kek,
    word32 kekSz,
    byte * keyID,
    word32 keyIdSz,
    void * timePtr,
    byte * otherOID,
    word32 otherOIDSz,
    byte * other,
    word32 otherSz,
    int options
)
```

Adds KEKRI recipient.

Parameters:

- **pkcs7** PKCS7 structure
- **keyWrapOID** Key wrap algorithm OID
- **kek** Key encryption key
- **kekSz** KEK size
- **keyID** Key identifier
- **keyIdSz** Key ID size
- **timePtr** Time pointer
- **otherOID** Other OID
- **otherOIDSz** Other OID size
- **other** Other data
- **otherSz** Other data size
- **options** Options flags

See: wc_PKCS7_AddRecipient_KTRI

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_AddRecipient_KEKRI(&pkcs7, AES256_WRAP, kek, kekSz,
                                      keyId, keyIdSz, NULL, NULL, 0,
                                      NULL, 0, 0);
```

C.41.4.58 function wc_PKCS7_SetPassword

```
int wc_PKCS7_SetPassword(
    wc_PKCS7 * pkcs7,
    byte * passwd,
    word32 pLen
)
```

Sets password for PWRI.

Parameters:

- **pkcs7** PKCS7 structure
- **passwd** Password
- **pLen** Password length

See: wc_PKCS7_AddRecipient_PWRI

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_SetPassword(&pkcs7, password, passwordLen);
```

C.41.4.59 function wc_PKCS7_AddRecipient_PWRI

```
int wc_PKCS7_AddRecipient_PWRI(
    wc_PKCS7 * pkcs7,
    byte * passwd,
    word32 pLen,
    byte * salt,
    word32 saltSz,
    int kdfOID,
    int prfOID,
    int iterations,
    int kekEncryptOID,
    int options
)
```

Adds PWRI recipient.

Parameters:

- **pkcs7** PKCS7 structure
- **passwd** Password
- **pLen** Password length
- **salt** Salt
- **saltSz** Salt size
- **kdfOID** KDF algorithm OID
- **prfOID** PRF algorithm OID
- **iterations** Iteration count
- **kekEncryptOID** KEK encryption algorithm OID
- **options** Options flags

See: wc_PKCS7_SetPassword

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_AddRecipient_PWRI(&pkcs7, password, passwordLen,
                                     salt, saltSz, PBKDF2_OID, HMACh,
                                     10000, AES256CBCb, 0);
```

C.41.4.60 function wc_PKCS7_SetOriEncryptCtx

```
int wc_PKCS7_SetOriEncryptCtx(
    wc_PKCS7 * pkcs7,
    void * ctx
)
```

Sets originator encryption context.

Parameters:

- **pkcs7** PKCS7 structure
- **ctx** Context pointer

See: wc_PKCS7_SetOriDecryptCtx

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_SetOriEncryptCtx(&pkcs7, myContext);
```

C.41.4.61 function wc_PKCS7_SetOriDecryptCtx

```
int wc_PKCS7_SetOriDecryptCtx(
    wc_PKCS7 * pkcs7,
    void * ctx
)
```

Sets originator decryption context.

Parameters:

- **pkcs7** PKCS7 structure
- **ctx** Context pointer

See: wc_PKCS7_SetOriEncryptCtx

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_SetOriDecryptCtx(&pkcs7, myContext);
```

C.41.4.62 function wc_PKCS7_SetOriDecryptCb

```
int wc_PKCS7_SetOriDecryptCb(
    wc_PKCS7 * pkcs7,
    CallbackOriDecrypt cb
)
```

Sets originator decryption callback.

Parameters:

- **pkcs7** PKCS7 structure
- **cb** Callback function

See: wc_PKCS7_SetOriDecryptCtx

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_SetOriDecryptCb(&pkcs7, myDecryptCallback);
```

C.41.4.63 function wc_PKCS7_AddRecipient_ORI

```
int wc_PKCS7_AddRecipient_ORI(
    wc_PKCS7 * pkcs7,
    CallbackOriEncrypt cb,
    int options
)
```

Adds ORI recipient.

Parameters:

- **pkcs7** PKCS7 structure
- **cb** Originator encryption callback
- **options** Options flags

See: wc_PKCS7_SetOriDecryptCb

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_AddRecipient_ORI(&pkcs7, myEncryptCallback, 0);
```

C.41.4.64 function wc_PKCS7_SetWrapCEKCb

```
int wc_PKCS7_SetWrapCEKCb(
    wc_PKCS7 * pkcs7,
    CallbackWrapCEK wrapCEKCb
)
```

Sets CEK wrap callback.

Parameters:

- **pkcs7** PKCS7 structure

- **wrapCEKCb** Wrap CEK callback

See: wc_PKCS7_Init

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_SetWrapCEKCb(&pkcs7, myWrapCEKCallback);
```

C.41.4.65 function wc_PKCS7_SetRsaSignRawDigestCb

```
int wc_PKCS7_SetRsaSignRawDigestCb(
    wc_PKCS7 * pkcs7,
    CallbackRsaSignRawDigest cb
)
```

Sets RSA sign raw digest callback.

Parameters:

- **pkcs7** PKCS7 structure
- **cb** Callback function

See: wc_PKCS7_Init

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_SetRsaSignRawDigestCb(&pkcs7, mySignCallback);
```

C.41.4.66 function wc_PKCS7_EncodeAuthEnvelopedData

```
int wc_PKCS7_EncodeAuthEnvelopedData(
    wc_PKCS7 * pkcs7,
    byte * output,
    word32 outputSz
)
```

Encodes authenticated enveloped data.

Parameters:

- **pkcs7** PKCS7 structure
- **output** Output buffer
- **outputSz** Output buffer size

See: wc_PKCS7_DecodeAuthEnvelopedData

Return:

- Size of encoded data on success
- negative on error

Example

```
int ret = wc_PKCS7_EncodeAuthEnvelopedData(&pkcs7, out, outSz);
```

C.41.4.67 function wc_PKCS7_DecodeAuthEnvelopedData

```
int wc_PKCS7_DecodeAuthEnvelopedData(
    wc_PKCS7 * pkcs7,
    byte * pkiMsg,
    word32 pkiMsgSz,
    byte * output,
    word32 outputSz
)
```

Decodes authenticated enveloped data.

Parameters:

- **pkcs7** PKCS7 structure
- **pkiMsg** Input message
- **pkiMsgSz** Input message size
- **output** Output buffer
- **outputSz** Output buffer size

See: wc_PKCS7_EncodeAuthEnvelopedData

Return:

- Size of decoded data on success
- negative on error

Example

```
int ret = wc_PKCS7_DecodeAuthEnvelopedData(&pkcs7, msg, msgSz, out,
                                           outSz);
```

C.41.4.68 function wc_PKCS7_EncodeEncryptedData

```
int wc_PKCS7_EncodeEncryptedData(
    wc_PKCS7 * pkcs7,
    byte * output,
    word32 outputSz
)
```

Encodes encrypted data.

Parameters:

- **pkcs7** PKCS7 structure
- **output** Output buffer
- **outputSz** Output buffer size

See: wc_PKCS7_DecodeEncryptedData

Return:

- Size of encoded data on success
- negative on error

Example

```
int ret = wc_PKCS7_EncodeEncryptedData(&pkcs7, out, outSz);
```

C.41.4.69 function wc_PKCS7_SetDecodeEncryptedCb

```
int wc_PKCS7_SetDecodeEncryptedCb(
    wc_PKCS7 * pkcs7,
    CallbackDecryptContent decryptionCb
)
```

Sets decode encrypted callback.

Parameters:

- **pkcs7** PKCS7 structure
- **decryptionCb** Decryption callback

See: wc_PKCS7_SetDecodeEncryptedCtx

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_SetDecodeEncryptedCb(&pkcs7, myDecryptCallback);
```

C.41.4.70 function wc_PKCS7_SetDecodeEncryptedCtx

```
int wc_PKCS7_SetDecodeEncryptedCtx(
    wc_PKCS7 * pkcs7,
    void * ctx
)
```

Sets decode encrypted context.

Parameters:

- **pkcs7** PKCS7 structure
- **ctx** Context pointer

See: wc_PKCS7_SetDecodeEncryptedCb

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_SetDecodeEncryptedCtx(&pkcs7, myContext);
```

C.41.4.71 function wc_PKCS7_SetStreamMode

```
int wc_PKCS7_SetStreamMode(
    wc_PKCS7 * pkcs7,
    byte flag,
    CallbackGetContent getContentCb,
    CallbackStreamOut streamOutCb,
    void * ctx
)
```

Sets stream mode for PKCS7.

Parameters:

- **pkcs7** PKCS7 structure
- **flag** Stream mode flag
- **getContentCb** Get content callback
- **streamOutCb** Stream output callback
- **ctx** Context pointer

See: wc_PKCS7_GetStreamMode

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_SetStreamMode(&pkcs7, 1, getContent, streamOut,
                                 ctx);
```

C.41.4.72 function wc_PKCS7_GetStreamMode

```
int wc_PKCS7_GetStreamMode(
    wc_PKCS7 * pkcs7
)
```

Gets stream mode setting.

Parameters:

- **pkcs7** PKCS7 structure

See: wc_PKCS7_SetStreamMode

Return: Stream mode flag

Example

```
int mode = wc_PKCS7_GetStreamMode(&pkcs7);
```

C.41.4.73 function wc_PKCS7_SetNoCerts

```
int wc_PKCS7_SetNoCerts(
    wc_PKCS7 * pkcs7,
    byte flag
)
```

Sets no certificates flag.

Parameters:

- **pkcs7** PKCS7 structure
- **flag** No certificates flag

See: wc_PKCS7_GetNoCerts

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_PKCS7_SetNoCerts(&pkcs7, 1);
```

C.41.4.74 function wc_PKCS7_GetNoCerts

```
int wc_PKCS7_GetNoCerts(
    wc_PKCS7 * pkcs7
)
```

Gets no certificates flag.

Parameters:

- **pkcs7** PKCS7 structure

See: wc_PKCS7_SetNoCerts

Return: No certificates flag

Example

```
int noCerts = wc_PKCS7_GetNoCerts(&pkcs7);
```

C.41.4.75 function wc_PKCS7_EncodeCompressedData

```
int wc_PKCS7_EncodeCompressedData(
    wc_PKCS7 * pkcs7,
    byte * output,
    word32 outputSz
)
```

Encodes compressed data.

Parameters:

- **pkcs7** PKCS7 structure
- **output** Output buffer
- **outputSz** Output buffer size

See: wc_PKCS7_DecodeCompressedData

Return:

- Size of encoded data on success
- negative on error

Example

```
int ret = wc_PKCS7_EncodeCompressedData(&pkcs7, out, outSz);
```

C.41.4.76 function wc_PKCS7_DecodeCompressedData

```
int wc_PKCS7_DecodeCompressedData(
    wc_PKCS7 * pkcs7,
    byte * pkiMsg,
    word32 pkiMsgSz,
    byte * output,
    word32 outputSz
)
```

Decodes compressed data.

Parameters:

- **pkcs7** PKCS7 structure
- **pkiMsg** Input message
- **pkiMsgSz** Input message size

- **output** Output buffer
- **outputSz** Output buffer size

See: wc_PKCS7_EncodeCompressedData

Return:

- Size of decoded data on success
- negative on error

Example

```
int ret = wc_PKCS7_DecodeCompressedData(&pkcs7, msg, msgSz, out,
                                        outSz);
```

C.41.5 Source code

```
typedef int (*CallbackAESKeyWrapUnwrap)(const byte* key, word32 keySz,
        const byte* in, word32 inSz, int wrap, byte* out, word32 outSz);

int  wc_PKCS7_InitWithCert(wc_PKCS7* pkcs7, byte* der, word32 derSz);

void wc_PKCS7_Free(wc_PKCS7* pkcs7);

int  wc_PKCS7_EncodeData(wc_PKCS7* pkcs7, byte* output,
                                      word32 outputSz);

int  wc_PKCS7_EncodeSignedData(wc_PKCS7* pkcs7,
                                      byte* output, word32 outputSz);

int wc_PKCS7_EncodeSignedData_ex(wc_PKCS7* pkcs7, const byte* hashBuf,
    word32 hashSz, byte* outputHead, word32* outputHeadSz, byte* outputFoot,
    word32* outputFootSz);

int  wc_PKCS7_VerifySignedData(wc_PKCS7* pkcs7,
                                      byte* pkiMsg, word32 pkiMsgSz);


int wc_PKCS7_VerifySignedData_ex(wc_PKCS7* pkcs7, const byte* hashBuf,
    word32 hashSz, byte* pkiMsgHead, word32 pkiMsgHeadSz, byte* pkiMsgFoot,
    word32 pkiMsgFootSz);

int wc_PKCS7_SetAESKeyWrapUnwrapCb(wc_PKCS7* pkcs7,
        CallbackAESKeyWrapUnwrap aesKeyWrapCb);

int  wc_PKCS7_EncodeEnvelopedData(wc_PKCS7* pkcs7,
                                        byte* output, word32 outputSz);

int wc_PKCS7_DecodeEnvelopedData(wc_PKCS7* pkcs7, byte* pkiMsg,
        word32 pkiMsgSz, byte* output, word32 outputSz);

int wc_PKCS7_GetEnvelopedDataKariRid(const byte * in, word32 inSz,
        byte * out, word32 * outSz);

int wc_PKCS7_DecodeEncryptedData(wc_PKCS7* pkcs7, byte* pkiMsg,
```

```
        word32 pkiMsgSz, byte* output, word32 outputSz);

int wc_PKCS7_DecodeEncryptedKeyPackage(wc_PKCS7 * pkcs7,
        byte * pkiMsg, word32 pkiMsgSz, byte * output, word32 outputSz);

int wc_PKCS7_DecodeSymmetricKeyPackageAttribute(const byte * skp,
        word32 skpSz, size_t index, const byte ** attr, word32 * attrSz);

int wc_PKCS7_DecodeSymmetricKeyPackageKey(const byte * skp,
        word32 skpSz, size_t index, const byte ** key, word32 * keySz);

int wc_PKCS7_DecodeOneSymmetricKeyAttribute(const byte * osk,
        word32 oskSz, size_t index, const byte ** attr, word32 * attrSz);

int wc_PKCS7_DecodeOneSymmetricKeyKey(const byte * osk,
        word32 oskSz, const byte ** key, word32 * keySz);

PKCS7* wolfSSL_PKCS7_new(void);

PKCS7_SIGNED* wolfSSL_PKCS7_SIGNED_new(void);

void wolfSSL_PKCS7_free(PKCS7* p7);

void wolfSSL_PKCS7_SIGNED_free(PKCS7_SIGNED* p7);

PKCS7* wolfSSL_d2i_PKCS7(PKCS7** p7, const unsigned char** in, int len);

PKCS7* wolfSSL_d2i_PKCS7_bio(WOLFSSL_BIO* bio, PKCS7** p7);

int wolfSSL_i2d_PKCS7_bio(WOLFSSL_BIO *bio, PKCS7 *p7);

int wolfSSL_i2d_PKCS7(PKCS7 *p7, unsigned char **out);

PKCS7* wolfSSL_PKCS7_sign(WOLFSSL_X509* signer, WOLFSSL_EVP_PKEY* pkey,
                          WOLFSSL_STACK* certs, WOLFSSL_BIO* in, int flags);

int wolfSSL_PKCS7_verify(PKCS7* p7, WOLFSSL_STACK* certs,
                         WOLFSSL_X509_STORE* store, WOLFSSL_BIO* in,
                         WOLFSSL_BIO* out, int flags);

int wolfSSL_PKCS7_final(PKCS7* pkcs7, WOLFSSL_BIO* in, int flags);

int wolfSSL_PKCS7_encode_certs(PKCS7* p7, WOLFSSL_STACK* certs,
                               WOLFSSL_BIO* out);

WOLFSSL_STACK* wolfSSL_PKCS7_to_stack(PKCS7* pkcs7);

WOLFSSL_STACK* wolfSSL_PKCS7_get0_signers(PKCS7* p7, WOLFSSL_STACK* certs,
                                          int flags);

int wolfSSL_PEM_write_bio_PKCS7(WOLFSSL_BIO* bio, PKCS7* p7);

PKCS7* wolfSSL_SMIME_read_PKCS7(WOLFSSL_BIO* in, WOLFSSL_BIO** bcont);
```

```
int wolfSSL_SMIME_write_PKCS7(WOLFSSL_BIO* out, PKCS7* pkcs7,
                              WOLFSSL_BIO* in, int flags);

wc_PKCS7* wc_PKCS7_New(void* heap, int devId);

void wc_PKCS7_SetUnknownExtCallback(wc_PKCS7* pkcs7,
                                    wc_UnknownExtCallback cb);

int wc_PKCS7_Init(wc_PKCS7* pkcs7, void* heap, int devId);

int wc_PKCS7_AddCertificate(wc_PKCS7* pkcs7, byte* der, word32 derSz);

int wc_PKCS7_GetAttributeValue(wc_PKCS7* pkcs7, const byte* oid,
                               word32 oidSz, byte* out, word32* outSz);

int wc_PKCS7_SetSignerIdentifierType(wc_PKCS7* pkcs7, int type);

int wc_PKCS7_SetContentType(wc_PKCS7* pkcs7, byte* contentType, word32 sz);

int wc_PKCS7_GetPadSize(word32 inputSz, word32 blockSz);

int wc_PKCS7_PadData(byte* in, word32 inSz, byte* out, word32 outSz,
                     word32 blockSz);

int wc_PKCS7_SetCustomSKID(wc_PKCS7* pkcs7, const byte* in, word16 inSz);

int wc_PKCS7_SetDetached(wc_PKCS7* pkcs7, word16 flag);

int wc_PKCS7_NoDefaultSignedAttribs(wc_PKCS7* pkcs7);

int wc_PKCS7_SetDefaultSignedAttribs(wc_PKCS7* pkcs7, word16 flag);

void wc_PKCS7_AllowDegenerate(wc_PKCS7* pkcs7, word16 flag);

int wc_PKCS7_GetSignerSID(wc_PKCS7* pkcs7, byte* out, word32* outSz);

int wc_PKCS7_EncodeSignedFPD(wc_PKCS7* pkcs7, byte* privateKey,
                             word32 privateKeySz, int signOID, int hashOID,
                             byte* content, word32 contentSz,
                             PKCS7Attrib* signedAttribs,
                             word32 signedAttribsSz, byte* output,
                             word32 outputSz);

int wc_PKCS7_EncodeSignedEncryptedFPD(wc_PKCS7* pkcs7, byte* encryptKey,
                                      word32 encryptKeySz, byte* privateKey,
                                      word32 privateKeySz, int encryptOID,
                                      int signOID, int hashOID, byte* content,
                                      word32 contentSz,
                                      PKCS7Attrib* unprotectedAttribs,
                                      word32 unprotectedAttribsSz,
                                      PKCS7Attrib* signedAttribs,
                                      word32 signedAttribsSz, byte* output,
                                      word32 outputSz);
```

```
int wc_PKCS7_EncodeSignedCompressedFPD(wc_PKCS7* pkcs7, byte* privateKey,
                                       word32 privateKeySz, int signOID,
                                       int hashOID, byte* content,
                                       word32 contentSz,
                                       PKCS7Attrib* signedAttribs,
                                       word32 signedAttribsSz, byte* output,
                                       word32 outputSz);

int wc_PKCS7_EncodeSignedEncryptedCompressedFPD(wc_PKCS7* pkcs7,
                                                byte* encryptKey,
                                                word32 encryptKeySz,
                                                byte* privateKey,
                                                word32 privateKeySz,
                                                int encryptOID, int signOID,
                                                int hashOID, byte* content,
                                                word32 contentSz,
                                               PKCS7Attrib* unprotectedAttribs,
                                                word32 unprotectedAttribsSz,
                                                PKCS7Attrib* signedAttribs,
                                                word32 signedAttribsSz,
                                                byte* output, word32 outputSz);

int wc_PKCS7_AddRecipient_KTRI(wc_PKCS7* pkcs7, const byte* cert,
                               word32 certSz, int options);

int wc_PKCS7_AddRecipient_KARI(wc_PKCS7* pkcs7, const byte* cert,
                               word32 certSz, int keyWrapOID,
                               int keyAgreeOID, byte* ukm, word32 ukmSz,
                               int options);

int wc_PKCS7_SetKey(wc_PKCS7* pkcs7, byte* key, word32 keySz);

int wc_PKCS7_AddRecipient_KEKRI(wc_PKCS7* pkcs7, int keyWrapOID, byte* kek,
                                word32 kekSz, byte* keyID, word32 keyIdSz,
                                void* timePtr, byte* otherOID,
                                word32 otherOIDSz, byte* other,
                                word32 otherSz, int options);

int wc_PKCS7_SetPassword(wc_PKCS7* pkcs7, byte* passwd, word32 pLen);

int wc_PKCS7_AddRecipient_PWRI(wc_PKCS7* pkcs7, byte* passwd, word32 pLen,
                               byte* salt, word32 saltSz, int kdfOID,
                               int prfOID, int iterations,
                               int kekEncryptOID, int options);

int wc_PKCS7_SetOriEncryptCtx(wc_PKCS7* pkcs7, void* ctx);

int wc_PKCS7_SetOriDecryptCtx(wc_PKCS7* pkcs7, void* ctx);

int wc_PKCS7_SetOriDecryptCb(wc_PKCS7* pkcs7, CallbackOriDecrypt cb);

int wc_PKCS7_AddRecipient_ORI(wc_PKCS7* pkcs7, CallbackOriEncrypt cb,
                              int options);
```

```
int wc_PKCS7_SetWrapCEKCb(wc_PKCS7* pkcs7, CallbackWrapCEK wrapCEKCb);

int wc_PKCS7_SetRsaSignRawDigestCb(wc_PKCS7* pkcs7,
                                   CallbackRsaSignRawDigest cb);

int wc_PKCS7_EncodeAuthEnvelopedData(wc_PKCS7* pkcs7, byte* output,
                                     word32 outputSz);

int wc_PKCS7_DecodeAuthEnvelopedData(wc_PKCS7* pkcs7, byte* pkiMsg,
                                     word32 pkiMsgSz, byte* output,
                                     word32 outputSz);

int wc_PKCS7_EncodeEncryptedData(wc_PKCS7* pkcs7, byte* output,
                                 word32 outputSz);

int wc_PKCS7_SetDecodeEncryptedCb(wc_PKCS7* pkcs7,
                                  CallbackDecryptContent decryptionCb);

int wc_PKCS7_SetDecodeEncryptedCtx(wc_PKCS7* pkcs7, void* ctx);

int wc_PKCS7_SetStreamMode(wc_PKCS7* pkcs7, byte flag,
                           CallbackGetContent getContentCb,
                           CallbackStreamOut streamOutCb, void* ctx);

int wc_PKCS7_GetStreamMode(wc_PKCS7* pkcs7);

int wc_PKCS7_SetNoCerts(wc_PKCS7* pkcs7, byte flag);

int wc_PKCS7_GetNoCerts(wc_PKCS7* pkcs7);

int wc_PKCS7_EncodeCompressedData(wc_PKCS7* pkcs7, byte* output,
                                  word32 outputSz);

int wc_PKCS7_DecodeCompressedData(wc_PKCS7* pkcs7, byte* pkiMsg,
                                  word32 pkiMsgSz, byte* output,
                                  word32 outputSz);
```

C.42 poly1305.h

C.42.1 Functions

	Name
int	**wc_Poly1305SetKey**(Poly1305 * poly1305, const byte * key, word32 kySz)This function sets the key for a Poly1305 context structure, initializing it for hashing. Note: A new key should be set after generating a message hash with wc_Poly1305Final to ensure security.
int	**wc_Poly1305Update**(Poly1305 * poly1305, const byte * m, word32 bytes)This function updates the message to hash with the Poly1305 structure.

	Name
int	**wc_Poly1305Final**(Poly1305 * poly1305, byte * tag)This function calculates the hash of the input messages and stores the result in mac. After this is called, the key should be reset.
int	**wc_Poly1305_MAC**(Poly1305 * ctx, const byte * additional, word32 addSz, const byte * input, word32 sz, byte * tag, word32 tagSz)Takes in an initialized Poly1305 struct that has a key loaded and creates a MAC (tag) using recent TLS AEAD padding scheme.
int	**wc_Poly1305_Pad**(Poly1305 * ctx, word32 lenToPad)Adds padding to Poly1305 context.
int	**wc_Poly1305_EncodeSizes**(Poly1305 * ctx, word32 aadSz, word32 dataSz)Encodes AAD and data sizes for Poly1305.
int	**wc_Poly1305_EncodeSizes64**(Poly1305 * ctx, word64 aadSz, word64 dataSz)Encodes AAD and data sizes for Poly1305 using 64-bit values.

C.42.2 Functions Documentation

C.42.2.1 function wc_Poly1305SetKey

```
int wc_Poly1305SetKey(
    Poly1305 * poly1305,
    const byte * key,
    word32 kySz
)
```

This function sets the key for a Poly1305 context structure, initializing it for hashing. Note: A new key should be set after generating a message hash with wc_Poly1305Final to ensure security.

Parameters:

- **ctx** pointer to a Poly1305 structure to initialize
- **key** pointer to the buffer containing the key to use for hashing
- **keySz** size of the key in the buffer. Should be 32 bytes

See:

- wc_Poly1305Update
- wc_Poly1305Final

Return:

- 0 Returned on successfully setting the key and initializing the Poly1305 structure
- BAD_FUNC_ARG Returned if the given key is not 32 bytes long, or the Poly1305 context is NULL

Example

```
Poly1305 enc;
byte key[] = { initialize with 32 byte key to use for hashing };
wc_Poly1305SetKey(&enc, key, sizeof(key));
```

C.42.2.2 function wc_Poly1305Update

```
int wc_Poly1305Update(
    Poly1305 * poly1305,
    const byte * m,
    word32 bytes
)
```

This function updates the message to hash with the Poly1305 structure.

Parameters:

- **ctx** pointer to a Poly1305 structure for which to update the message to hash
- **m** pointer to the buffer containing the message which should be added to the hash
- **bytes** size of the message to hash

See:

- wc_Poly1305SetKey
- wc_Poly1305Final

Return:

- 0 Returned on successfully updating the message to hash
- BAD_FUNC_ARG Returned if the Poly1305 structure is NULL

Example

```
Poly1305 enc;
byte key[] = { }; // initialize with 32 byte key to use for encryption

byte msg[] = { }; // initialize with message to hash
wc_Poly1305SetKey(&enc, key, sizeof(key));

if( wc_Poly1305Update(key, msg, sizeof(msg)) != 0 ) {
    // error updating message to hash
}
```

C.42.2.3 function wc_Poly1305Final

```
int wc_Poly1305Final(
    Poly1305 * poly1305,
    byte * tag
)
```

This function calculates the hash of the input messages and stores the result in mac. After this is called, the key should be reset.

Parameters:

- **ctx** pointer to a Poly1305 structure with which to generate the MAC
- **mac** pointer to the buffer in which to store the MAC. Should be POLY1305_DIGEST_SIZE (16 bytes) wide

See:

- wc_Poly1305SetKey
- wc_Poly1305Update

Return:

- 0 Returned on successfully computing the final MAC
- BAD_FUNC_ARG Returned if the Poly1305 structure is NULL

Example

```
Poly1305 enc;
byte mac[POLY1305_DIGEST_SIZE]; // space for a 16 byte mac

byte key[] = { }; // initialize with 32 byte key to use for encryption

byte msg[] = { }; // initialize with message to hash
wc_Poly1305SetKey(&enc, key, sizeof(key));
wc_Poly1305Update(key, msg, sizeof(msg));

if ( wc_Poly1305Final(&enc, mac) != 0 ) {
    // error computing final MAC
}
```

C.42.2.4 function wc_Poly1305_MAC

```
int wc_Poly1305_MAC(
    Poly1305 * ctx,
    const byte * additional,
    word32 addSz,
    const byte * input,
    word32 sz,
    byte * tag,
    word32 tagSz
)
```

Takes in an initialized Poly1305 struct that has a key loaded and creates a MAC (tag) using recent TLS AEAD padding scheme.

Parameters:

- **ctx** Initialized Poly1305 struct to use
- **additional** Additional data to use
- **addSz** Size of additional buffer
- **input** Input buffer to create tag from
- **sz** Size of input buffer
- **tag** Buffer to hold created tag
- **tagSz** Size of input tag buffer (must be at least WC_POLY1305_MAC_SZ(16))

See:

- wc_Poly1305SetKey
- wc_Poly1305Update
- wcPoly1305Final

Return:

- 0 Success
- BAD_FUNC_ARG Returned if ctx, input, or tag is null or if additional is null and addSz is greater than 0 or if tagSz is less than WC_POLY1305_MAC_SZ.

Example

```
Poly1305 ctx;
byte key[] = { }; // initialize with 32 byte key to use for hashing
byte additional[] = { }; // initialize with additional data
byte msg[] = { }; // initialize with message
byte tag[16];

wc_Poly1305SetKey(&ctx, key, sizeof(key));
```

```
if(wc_Poly1305_MAC(&ctx, additional, sizeof(additional), (byte*)msg,
sizeof(msg), tag, sizeof(tag)) != 0)
{
    // Handle the error
}
```

C.42.2.5 function wc_Poly1305_Pad

```
int wc_Poly1305_Pad(
    Poly1305 * ctx,
    word32 lenToPad
)
```

Adds padding to Poly1305 context.

Parameters:

- **ctx** Poly1305 context
- **lenToPad** Length to pad

See: wc_Poly1305_MAC

Return:

- 0 on success
- BAD_FUNC_ARG if ctx is NULL

Example

```
Poly1305 ctx;
byte key[32];
wc_Poly1305SetKey(&ctx, key, sizeof(key));
int ret = wc_Poly1305_Pad(&ctx, 10);
```

C.42.2.6 function wc_Poly1305_EncodeSizes

```
int wc_Poly1305_EncodeSizes(
    Poly1305 * ctx,
    word32 aadSz,
    word32 dataSz
)
```

Encodes AAD and data sizes for Poly1305.

Parameters:

- **ctx** Poly1305 context
- **aadSz** Additional authenticated data size
- **dataSz** Data size

See: wc_Poly1305_MAC

Return:

- 0 on success
- BAD_FUNC_ARG if ctx is NULL

Example

```
Poly1305 ctx;
byte key[32];
```

```
wc_Poly1305SetKey(&ctx, key, sizeof(key));
int ret = wc_Poly1305_EncodeSizes(&ctx, 16, 100);
```

C.42.2.7 function wc_Poly1305_EncodeSizes64

```
int wc_Poly1305_EncodeSizes64(
    Poly1305 * ctx,
    word64 aadSz,
    word64 dataSz
)
```

Encodes AAD and data sizes for Poly1305 using 64-bit values.

Parameters:

- **ctx** Poly1305 context
- **aadSz** Additional authenticated data size
- **dataSz** Data size

See: wc_Poly1305_EncodeSizes

Return:

- 0 on success
- BAD_FUNC_ARG if ctx is NULL

Example

```
Poly1305 ctx;
byte key[32];
wc_Poly1305SetKey(&ctx, key, sizeof(key));
int ret = wc_Poly1305_EncodeSizes64(&ctx, 16, 100);
```

C.42.3 Source code

```
int wc_Poly1305SetKey(Poly1305* poly1305, const byte* key,
                                  word32 kySz);

int wc_Poly1305Update(Poly1305* poly1305, const byte* m, word32 bytes);

int wc_Poly1305Final(Poly1305* poly1305, byte* tag);

int wc_Poly1305_MAC(Poly1305* ctx, const byte* additional, word32 addSz,
                    const byte* input, word32 sz, byte* tag, word32 tagSz);

int wc_Poly1305_Pad(Poly1305* ctx, word32 lenToPad);

int wc_Poly1305_EncodeSizes(Poly1305* ctx, word32 aadSz, word32 dataSz);

int wc_Poly1305_EncodeSizes64(Poly1305* ctx, word64 aadSz, word64 dataSz);
```

C.43 psa.h

C.43.1 Functions

	Name
int	**wolfSSL_CTX_psa_enable**(WOLFSSL_CTX * ctx)This function enables PSA support on the given context.
int	**wolfSSL_set_psa_ctx**(WOLFSSL * ssl, struct psa_ssl_ctx * ctx)This function setup the PSA context for the given SSL session.
void	**wolfSSL_free_psa_ctx**(struct psa_ssl_ctx * ctx)This function releases the resources used by a PSA context.
int	**wolfSSL_psa_set_private_key_id**(struct psa_ssl_ctx * ctx, psa_key_id_t id)This function set the private key used by an SSL session.
int	**wc_psa_get_random**(unsigned char * out, word32 sz)This function generates random bytes using the PSA crypto API. This is a wrapper around the PSA random number generation functions.
int	**wc_psa_aes_encrypt_decrypt**(Aes * aes, const uint8_t * input, uint8_t * output, size_t length, psa_algorithm_t alg, int direction)This function performs AES encryption or decryption using the PSA crypto API. It supports various AES modes through the algorithm parameter.

C.43.2 Functions Documentation

C.43.2.1 function wolfSSL_CTX_psa_enable

```
int wolfSSL_CTX_psa_enable(
    WOLFSSL_CTX * ctx
)
```

This function enables PSA support on the given context.

Parameters:

- **ctx** pointer to the WOLFSSL_CTX object on which the PSA support must be enabled

See: wolfSSL_set_psa_ctx

Return:

- WOLFSSL_SUCCESS on success
- BAD_FUNC_ARG if ctx == NULL

Example

```
WOLFSSL_CTX *ctx;
ctx = wolfSSL_CTX_new(wolfTLSv1_2_client_method());
if (!ctx)
    return NULL;
ret = wolfSSL_CTX_psa_enable(ctx);
if (ret != WOLFSSL_SUCCESS)
    printf("can't enable PSA on ctx");
```

C.43.2.2 function wolfSSL_set_psa_ctx

```
int wolfSSL_set_psa_ctx(
    WOLFSSL * ssl,
    struct psa_ssl_ctx * ctx
)
```

This function setup the PSA context for the given SSL session.

Parameters:

- **ssl** pointer to the WOLFSSL where the ctx will be enabled
- **ctx** pointer to a struct psa_ssl_ctx (must be unique for a ssl session)

See:

- wolfSSL_psa_set_private_key_id
- wolfSSL_psa_free_psa_ctx

Return:

- WOLFSSL_SUCCESS on success
- BAD_FUNC_ARG if ssl or ctx are NULL

This function setup the PSA context for the TLS callbacks to the given SSL session. At the end of the session, the resources used by the context should be freed using wolfSSL_free_psa_ctx().

Example

```
// Create new ssl session
WOLFSSL *ssl;
struct psa_ssl_ctx psa_ctx = { 0 };
ssl = wolfSSL_new(ctx);
if (!ssl)
    return NULL;
// setup PSA context
ret = wolfSSL_set_psa_ctx(ssl, ctx);
```

C.43.2.3 function wolfSSL_free_psa_ctx

```
void wolfSSL_free_psa_ctx(
    struct psa_ssl_ctx * ctx
)
```

This function releases the resources used by a PSA context.

Parameters:

- **ctx** pointer to a struct psa_ssl_ctx

See: wolfSSL_set_psa_ctx

C.43.2.4 function wolfSSL_psa_set_private_key_id

```
int wolfSSL_psa_set_private_key_id(
    struct psa_ssl_ctx * ctx,
    psa_key_id_t id
)
```

This function set the private key used by an SSL session.

Parameters:

- **ctx** pointer to a struct psa_ssl_ctx
- **id** PSA id of the key to be used as private key

See: wolfSSL_set_psa_ctx

Example

```
// Create new ssl session
WOLFSSL *ssl;
struct psa_ssl_ctx psa_ctx = { 0 };
psa_key_id_t key_id;

// key provisioning already done
get_private_key_id(&key_id);

ssl = wolfSSL_new(ctx);
if (!ssl)
    return NULL;

wolfSSL_psa_set_private_key_id(&psa_ctx, key_id);
wolfSSL_set_psa_ctx(ssl, ctx);
```

C.43.2.5 function wc_psa_get_random

```
int wc_psa_get_random(
    unsigned char * out,
    word32 sz
)
```

This function generates random bytes using the PSA crypto API. This is a wrapper around the PSA random number generation functions.

Parameters:

- **out** pointer to buffer to store random bytes
- **sz** number of random bytes to generate

See: wc_RNG_GenerateBlock

Return:

- 0 On success
- Negative value on error

Example

```
byte random[32];

int ret = wc_psa_get_random(random, sizeof(random));
if (ret != 0) {
    // error generating random bytes
}
```

C.43.2.6 function wc_psa_aes_encrypt_decrypt

```
int wc_psa_aes_encrypt_decrypt(
    Aes * aes,
    const uint8_t * input,
    uint8_t * output,
    size_t length,
```

```
    psa_algorithm_t alg,
    int direction
)
```

This function performs AES encryption or decryption using the PSA crypto API. It supports various AES modes through the algorithm parameter.

Parameters:

- **aes** pointer to initialized Aes structure
- **input** pointer to input data buffer
- **output** pointer to output data buffer
- **length** length of data to process
- **alg** PSA algorithm identifier specifying the AES mode
- **direction** encryption (1) or decryption (0)

See:

- wc_AesEncrypt
- wc_AesDecrypt

Return:

- 0 On success
- Negative value on error

Example

```
Aes aes;
byte key[16] = { }; // AES key
byte input[16] = { }; // plaintext
byte output[16];

wc_AesInit(&aes, NULL, INVALID_DEVID);
wc_AesSetKey(&aes, key, sizeof(key), NULL, AES_ENCRYPTION);
int ret = wc_psa_aes_encrypt_decrypt(&aes, input, output,
                                     sizeof(input),
                                     PSA_ALG_ECB_NO_PADDING, 1);
```

C.43.3 Source code

```
int wolfSSL_CTX_psa_enable(WOLFSSL_CTX *ctx);

int wolfSSL_set_psa_ctx(WOLFSSL *ssl, struct psa_ssl_ctx *ctx);

void wolfSSL_free_psa_ctx(struct psa_ssl_ctx *ctx);

int wolfSSL_psa_set_private_key_id(struct psa_ssl_ctx *ctx,
                                                psa_key_id_t id);

int wc_psa_get_random(unsigned char *out, word32 sz);

int wc_psa_aes_encrypt_decrypt(Aes *aes, const uint8_t *input,
                               uint8_t *output, size_t length,
                               psa_algorithm_t alg, int direction);
```

C.44 puf.h

C.44.1 Functions

	Name
int	**wc_PufInit**(wc_PufCtx * ctx)Initialize a wc_PufCtx structure, zeroing all fields. Must be called before any other PUF operations.
int	**wc_PufReadSram**(wc_PufCtx * ctx, const byte * sramAddr, word32 sramSz)Read raw SRAM data into the PUF context. The sramAddr should point to a NOLOAD linker section to preserve the power_on state. The required size, WC_PUF_RAW_BYTES, scales with WC_PUF_NUM_CODEWORDS (256 bytes at the default 16 codewords).
int	**wc_PufCheckSram**(const byte * sramAddr, word32 sramSz, word32 * onesCount)Health test a candidate raw SRAM readout without loading it into a context. Rejects a readout that cannot be SRAM power_on noise: any 128_bit block that is all zero or all ones, any block that repeats the block before it, or a total Hamming weight outside the WC_PUF_HW_MIN_PCT to WC_PUF_HW_MAX_PCT band (default 35% to 65% of WC_PUF_RAW_BITS).
int	**wc_PufEnroll**(wc_PufCtx * ctx)Perform PUF enrollment. Encodes raw SRAM using the selected BCH(127,k,t) profile (WC_PUF_BCH_T) and generates public helper data (WC_PUF_HELPER_BYTES). After enrollment the context is ready for key derivation and identity retrieval.
int	**wc_PufReconstruct**(wc_PufCtx * ctx, const byte * helperData, word32 helperSz)Reconstruct stable PUF bits from noisy SRAM using stored helper data. BCH error correction corrects up to WC_PUF_BCH_T bit flips per 127-bit codeword. The helper data and build configuration must match the enrollment that produced them.
int	**wc_PufDeriveKey**(wc_PufCtx * ctx, const byte * info, word32 infoSz, byte * key, word32 keySz)Derive a cryptographic key from PUF stable bits using HKDF. Uses SHA-256 by default, or SHA3-256 when WC_PUF_SHA3 is defined. The info parameter provides domain separation for multiple keys. Requires HAVE_HKDF.
int	**wc_PufGetIdentity**(wc_PufCtx * ctx, byte * id, word32 idSz)Retrieve the device identity hash (SHA_256 or SHA3_256 of stable bits). Deterministic for a given device.

	Name
int	**wc_PufGetParams**(int * m, int * n, int * k, int * t, int * numCodewords)Report the compile_time PUF profile parameters: field size m, codeword length n, message length k, error_correction capability t, and the number of codewords. Each output pointer is optional (may be NULL), but an all_NULL call is treated as a usage error. Enrollment and reconstruction firmware must agree on all of these (and the hash); persist WC_PUF_PROFILE_ID (which also encodes the hash selection) with the helper data and compare before reconstruction to detect a build mismatch.
word32	**wc_PufGetProfileId**(void)Return the profile fingerprint the LIBRARY was compiled with. The WC_PUF_PROFILE_ID macro necessarily reflects the including application's own build, so comparing the two detects a library/application mismatch - including a differing hash selection - that no length check can see.
int	**wc_PufGetHelperData**(wc_PufCtx * ctx, byte * helper, word32 helperSz)Copy out the enrollment helper data. Use this rather than reading wc_PufCtx.helperData directly: the size varies with the selected profile and with WC_PUF_HELPER_COMPACT.
int	**wc_PufReconstructEx**(wc_PufCtx * ctx, const byte * helperData, word32 helperSz, word32 profileId)Reconstruct as wc_PufReconstruct, but first check the caller's stored profile id against the library's. Helper-data size does not depend on t in the default layout, so the length check alone cannot detect helper data enrolled by a differently configured build - which would otherwise decode to a silently wrong key. Persist WC_PUF_PROFILE_ID alongside the helper data at enrollment and pass it back here.
int	**wc_PufZeroize**(wc_PufCtx * ctx)Securely zeroize all sensitive data in the PUF context using ForceZero. Call when PUF is no longer needed.
int	**wc_PufSetTestData**(wc_PufCtx * ctx, const byte * data, word32 sz)Inject synthetic SRAM test data for testing without hardware. Only available when WOLFSSL_PUF_TEST is defined.

C.44.2 Functions Documentation

C.44.2.1 function wc_PufInit

```
int wc_PufInit(
    wc_PufCtx * ctx
)
```

Initialize a wc_PufCtx structure, zeroing all fields. Must be called before any other PUF operations.

Parameters:

- **ctx** pointer to wc_PufCtx structure to initialize

See:

- wc_PufReadSram
- wc_PufEnroll
- wc_PufZeroize

Return:

- 0 on success
- BAD_FUNC_ARG if ctx is NULL

The SRAM PUF uses a configurable BCH(127,k,t) fuzzy extractor over GF(2^7) with HKDF key derivation. WC_PUF_BCH_T selects the error-correction strength (t=7, 10 default, 13, or 15) and WC_PUF_NUM_CODEWORDS (default 16) trades SRAM footprint and helper-data size (WC_PUF_HELPER_BYTES) against derived-key entropy. Only the (n - k) parity bits per codeword carry information; the leading k bits of each helper codeword are identically zero, so size OTP/flash accordingly. WC_PUF_HELPER_COMPACT stores only those parity bits, shrinking helper data to 39-72% of the default. It changes the stored format and so is opt-in; the default layout stays compatible with helper data enrolled by wolfSSL 5.9.2.

Enrollment and reconstruction must use identical WC_PUF_BCH_T and WC_PUF_NUM_CODEWORDS (and the same hash); persist WC_PUF_PROFILE_ID with the helper data at enrollment and pass it to wc_PufReconstructEx, or compare it against wc_PufGetProfileId(), to catch a build mismatch.

Every readout handed to wc_PufReadSram() is health tested first (see wc_PufCheckSram). A degenerate readout _ all zero, all ones, or a repeating block _ would otherwise pass cleanly through encoding, masking, decoding and HKDF and derive a key that is the same on every device. WC_PUF_HW_MIN_PCT and WC_PUF_HW_MAX_PCT (default 35 and 65) set the Hamming-weight band the readout must fall inside.

The test rejects degenerate readouts, not every already-written region: ordinary firmware content - .data copied from flash, a string table, a previous boot stage - is identical on every device yet neither constant nor strongly biased, so it can pass. Sampling the region from reset, before .bss/.data initialization and before it is used as stack or heap, remains a requirement of correct NOLOAD placement rather than something this test can enforce.

For a complete bare-metal example (tested on NUCLEO-H563ZI), see https://github.com/wolfSSL/wolfssl-examples/tree/master/puf

Example

```
wc_PufCtx ctx;
ret = wc_PufInit(&ctx);
```

C.44.2.2 function wc_PufReadSram

```
int wc_PufReadSram(
    wc_PufCtx * ctx,
    const byte * sramAddr,
    word32 sramSz
)
```

Read raw SRAM data into the PUF context. The sramAddr should point to a NOLOAD linker section to preserve the power-on state. The required size, WC_PUF_RAW_BYTES, scales with WC_PUF_NUM_CODEWORDS (256 bytes at the default 16 codewords).

Parameters:

- **ctx** pointer to wc_PufCtx structure
- **sramAddr** pointer to raw SRAM memory region
- **sramSz** size of SRAM buffer (must be >= WC_PUF_RAW_BYTES)

See:

- wc_PufInit
- wc_PufCheckSram
- wc_PufEnroll
- wc_PufReconstruct

Return:

- 0 on success
- BAD_FUNC_ARG if ctx or sramAddr is NULL
- PUF_READ_E if sramSz < WC_PUF_RAW_BYTES, or if the readout fails the health test

The readout is health tested with wc_PufCheckSram() before it is accepted. A degenerate readout _ typically a region already cleared by .bss init, the common bring_up mistake of sampling after C runtime startup _ is rejected with PUF_READ_E, and the context is left unusable by wc_PufEnroll() and wc_PufReconstruct(). The test catches degenerate shapes, not every already-written region, so it does not remove the requirement to sample from reset.

Example

```
__attribute__((section(".puf_sram")))
static volatile uint8_t puf_sram[WC_PUF_RAW_BYTES];
wc_PufReadSram(&ctx, (const byte*)puf_sram, sizeof(puf_sram));
```

C.44.2.3 function wc_PufCheckSram

```
int wc_PufCheckSram(
    const byte * sramAddr,
    word32 sramSz,
    word32 * onesCount
)
```

Health test a candidate raw SRAM readout without loading it into a context. Rejects a readout that cannot be SRAM power-on noise: any 128-bit block that is all zero or all ones, any block that repeats the block before it, or a total Hamming weight outside the WC_PUF_HW_MIN_PCT to WC_PUF_HW_MAX_PCT band (default 35% to 65% of WC_PUF_RAW_BITS).

Parameters:

- **sramAddr** pointer to raw SRAM memory region
- **sramSz** size of SRAM buffer (must be >= WC_PUF_RAW_BYTES)
- **onesCount** optional; receives the number of one bits in the first WC_PUF_RAW_BYTES of the readout, out of WC_PUF_RAW_BITS. Written whenever sramAddr is non-NULL and sramSz is large enough, whatever the verdict. It is a property of the raw PUF material, so treat it as a bring-up measurement and do not report it from production firmware

See:

- wc_PufReadSram
- wc_PufEnroll

Return:

- 0 if the readout is plausible PUF material
- BAD_FUNC_ARG if sramAddr is NULL
- PUF_READ_E if sramSz < WC_PUF_RAW_BYTES, or if the readout fails any of the checks

wc_PufReadSram() applies this test to every readout, so calling it directly is only needed to qualify a candidate SRAM region during board bring-up, or to report why a read was refused. onesCount is optional and is written whenever the size check passes - including when the readout is then rejected, so the measured bias of a rejected region is still available. It is left untouched when sramAddr is NULL or sramSz is short.

Example

```
word32 ones = 0;
ret = wc_PufCheckSram((const byte*)puf_sram, sizeof(puf_sram), &ones);
printf("PUF SRAM bias %u/%u ones, ret %d\n",
       (unsigned)ones, (unsigned)WC_PUF_RAW_BITS, ret);
```

C.44.2.4 function wc_PufEnroll

```
int wc_PufEnroll(
    wc_PufCtx * ctx
)
```

Perform PUF enrollment. Encodes raw SRAM using the selected BCH(127,k,t) profile (WC_PUF_BCH_T) and generates public helper data (WC_PUF_HELPER_BYTES). After enrollment the context is ready for key derivation and identity retrieval.

Parameters:

- **ctx** pointer to wc_PufCtx (must have SRAM data loaded)

See:

- wc_PufReadSram
- wc_PufReconstruct
- wc_PufDeriveKey

Return:

- 0 on success
- BAD_FUNC_ARG if ctx is NULL
- PUF_ENROLL_E if enrollment fails

Example

```
wc_PufEnroll(&ctx);
XMEMCPY(helperData, ctx.helperData, WC_PUF_HELPER_BYTES);
```

C.44.2.5 function wc_PufReconstruct

```
int wc_PufReconstruct(
    wc_PufCtx * ctx,
    const byte * helperData,
    word32 helperSz
)
```

Reconstruct stable PUF bits from noisy SRAM using stored helper data. BCH error correction corrects up to WC_PUF_BCH_T bit flips per 127-bit codeword. The helper data and build configuration must match the enrollment that produced them.

Parameters:

- **ctx** pointer to wc_PufCtx (must have SRAM data loaded)
- **helperData** pointer to helper data from previous enrollment
- **helperSz** size of helper data (>= WC_PUF_HELPER_BYTES)

See:

- wc_PufEnroll
- wc_PufDeriveKey
- wc_PufGetIdentity

Return:

- 0 on success
- BAD_FUNC_ARG if ctx or helperData is NULL
- PUF_RECONSTRUCT_E on failure (too many bit errors or helperSz too small)

Example

```
wc_PufReconstruct(&ctx, helperData, sizeof(helperData));
```

C.44.2.6 function wc_PufDeriveKey

```
int wc_PufDeriveKey(
    wc_PufCtx * ctx,
    const byte * info,
    word32 infoSz,
    byte * key,
    word32 keySz
)
```

Derive a cryptographic key from PUF stable bits using HKDF. Uses SHA-256 by default, or SHA3-256 when WC_PUF_SHA3 is defined. The info parameter provides domain separation for multiple keys. Requires HAVE_HKDF.

Parameters:

- **ctx** pointer to wc_PufCtx (must be enrolled or reconstructed)
- **info** optional context info for domain separation (may be NULL; when NULL, infoSz is treated as 0)
- **infoSz** size of info in bytes
- **key** output buffer for derived key
- **keySz** desired key size in bytes

See:

- wc_PufEnroll
- wc_PufReconstruct
- wc_PufGetIdentity

Return:

- 0 on success
- BAD_FUNC_ARG if ctx or key is NULL, or keySz is 0
- PUF_DERIVE_KEY_E if PUF not ready or HKDF fails

Example

```
byte key[32];
const byte info[] = "my-app-key";
wc_PufDeriveKey(&ctx, info, sizeof(info), key, sizeof(key));
```

C.44.2.7 function wc_PufGetIdentity

```
int wc_PufGetIdentity(
    wc_PufCtx * ctx,
    byte * id,
    word32 idSz
)
```

Retrieve the device identity hash (SHA-256 or SHA3-256 of stable bits). Deterministic for a given device.

Parameters:

- **ctx** pointer to wc_PufCtx (must be enrolled or reconstructed)
- **id** output buffer for identity hash
- **idSz** size of id buffer (>= WC_PUF_ID_SZ, 32 bytes)

See:

- wc_PufEnroll
- wc_PufReconstruct
- wc_PufDeriveKey

Return:

- 0 on success
- BAD_FUNC_ARG if ctx or id is NULL
- PUF_IDENTITY_E if PUF not ready or idSz < WC_PUF_ID_SZ

Example

```
byte identity[WC_PUF_ID_SZ];
wc_PufGetIdentity(&ctx, identity, sizeof(identity));
```

C.44.2.8 function wc_PufGetParams

```
int wc_PufGetParams(
    int * m,
    int * n,
    int * k,
    int * t,
    int * numCodewords
)
```

Report the compile-time PUF profile parameters: field size m, codeword length n, message length k, error-correction capability t, and the number of codewords. Each output pointer is optional (may be NULL), but an all-NULL call is treated as a usage error. Enrollment and reconstruction firmware must agree on all of these (and the hash); persist WC_PUF_PROFILE_ID (which also encodes the hash selection) with the helper data and compare before reconstruction to detect a build mismatch.

Parameters:

- **m** optional output for the GF field exponent (7 for GF(2^7))
- **n** optional output for the codeword length (127)
- **k** optional output for the message length (per WC_PUF_BCH_T)
- **t** optional output for the error-correction capability (WC_PUF_BCH_T)
- **numCodewords** optional output for WC_PUF_NUM_CODEWORDS

See:

- wc_PufEnroll
- wc_PufReconstruct

Return:

- 0 on success
- BAD_FUNC_ARG if every output pointer is NULL

Example

```
int t, numCodewords;
wc_PufGetParams(NULL, NULL, NULL, &t, &numCodewords);
```

C.44.2.9 function wc_PufGetProfileId

```
word32 wc_PufGetProfileId(
    void
)
```

Return the profile fingerprint the LIBRARY was compiled with. The WC_PUF_PROFILE_ID macro necessarily reflects the including application's own build, so comparing the two detects a library/application mismatch - including a differing hash selection - that no length check can see.

See:

- wc_PufGetParams
- wc_PufReconstructEx

Return: the library's WC_PUF_PROFILE_ID

Example

```
if (wc_PufGetProfileId() != WC_PUF_PROFILE_ID) {
    /* library and application were built with different PUF settings */
}
```

C.44.2.10 function wc_PufGetHelperData

```
int wc_PufGetHelperData(
    wc_PufCtx * ctx,
    byte * helper,
    word32 helperSz
)
```

Copy out the enrollment helper data. Use this rather than reading wc_PufCtx.helperData directly: the size varies with the selected profile and with WC_PUF_HELPER_COMPACT.

Parameters:

- **ctx** pointer to an enrolled wc_PufCtx
- **helper** output buffer for the helper data
- **helperSz** size of helper in bytes (>= WC_PUF_HELPER_BYTES)

See:

- wc_PufEnroll
- wc_PufReconstructEx

Return:

- 0 on success
- BAD_FUNC_ARG if ctx or helper is NULL
- PUF_ENROLL_E if the context is not enrolled, or helperSz is less than WC_PUF_HELPER_BYTES

Example

```
byte helper[WC_PUF_HELPER_BYTES];
wc_PufGetHelperData(&ctx, helper, sizeof(helper));
```

C.44.2.11 function wc_PufReconstructEx

```
int wc_PufReconstructEx(
    wc_PufCtx * ctx,
    const byte * helperData,
    word32 helperSz,
    word32 profileId
)
```

Reconstruct as wc_PufReconstruct, but first check the caller's stored profile id against the library's. Helper-data size does not depend on t in the default layout, so the length check alone cannot detect helper data enrolled by a differently configured build - which would otherwise decode to a silently wrong key. Persist WC_PUF_PROFILE_ID alongside the helper data at enrollment and pass it back here.

Parameters:

- **ctx** pointer to wc_PufCtx (must have SRAM data loaded)
- **helperData** pointer to helper data from a previous enrollment
- **helperSz** size of helper data (>= WC_PUF_HELPER_BYTES)
- **profileId** the WC_PUF_PROFILE_ID recorded at enrollment

See:

- wc_PufReconstruct
- wc_PufGetProfileId

Return:

- 0 on success
- BAD_FUNC_ARG if ctx or helperData is NULL, or profileId does not match the library's profile
- PUF_RECONSTRUCT_E on failure (too many bit errors or helperSz too small)

Example

```
wc_PufReconstructEx(&ctx, helper, sizeof(helper), savedProfileId);
```

C.44.2.12 function wc_PufZeroize

```
int wc_PufZeroize(
    wc_PufCtx * ctx
)
```

Securely zeroize all sensitive data in the PUF context using ForceZero. Call when PUF is no longer needed.

Parameters:

- **ctx** pointer to wc_PufCtx to zeroize

See: wc_PufInit

Return:

- 0 on success
- BAD_FUNC_ARG if ctx is NULL

Example

```
wc_PufZeroize(&ctx);
```

C.44.2.13 function wc_PufSetTestData

```
int wc_PufSetTestData(
    wc_PufCtx * ctx,
    const byte * data,
    word32 sz
)
```

Inject synthetic SRAM test data for testing without hardware. Only available when WOLFSSL_PUF_TEST is defined.

Parameters:

- **ctx** pointer to wc_PufCtx
- **data** pointer to synthetic SRAM data
- **sz** size of data (>= WC_PUF_RAW_BYTES)

See:

- wc_PufInit
- wc_PufReadSram

Return:

- 0 on success
- BAD_FUNC_ARG if ctx or data is NULL
- PUF_READ_E if sz < WC_PUF_RAW_BYTES

Example

```
byte testSram[WC_PUF_RAW_BYTES];
wc_PufSetTestData(&ctx, testSram, sizeof(testSram));
```

C.44.3 Source code

```
int wc_PufInit(wc_PufCtx* ctx);

int wc_PufReadSram(wc_PufCtx* ctx, const byte* sramAddr, word32 sramSz);

int wc_PufCheckSram(const byte* sramAddr, word32 sramSz, word32* onesCount);

int wc_PufEnroll(wc_PufCtx* ctx);

int wc_PufReconstruct(wc_PufCtx* ctx, const byte* helperData, word32 helperSz);

int wc_PufDeriveKey(wc_PufCtx* ctx, const byte* info, word32 infoSz,
                    byte* key, word32 keySz);

int wc_PufGetIdentity(wc_PufCtx* ctx, byte* id, word32 idSz);

int wc_PufGetParams(int* m, int* n, int* k, int* t, int* numCodewords);

    }
    \endcode

    \sa wc_PufGetParams
    \sa wc_PufReconstructEx
*/
```

```
word32 wc_PufGetProfileId(void);

int wc_PufGetHelperData(wc_PufCtx* ctx, byte* helper, word32 helperSz);

int wc_PufReconstructEx(wc_PufCtx* ctx, const byte* helperData,
                        word32 helperSz, word32 profileId);

int wc_PufZeroize(wc_PufCtx* ctx);

int wc_PufSetTestData(wc_PufCtx* ctx, const byte* data, word32 sz);
```

C.45 pwdbased.h

C.45.1 Functions

	Name
int	**wc_PBKDF1**(byte * output, const byte * passwd, int pLen, const byte * salt, int sLen, int iterations, int kLen, int hashType)This function implements the Password Based Key Derivation Function 1 (PBKDF1), converting an input password with a concatenated salt into a more secure key, which it stores in output. It allows the user to select between SHA and MD5 as hash functions.
int	**wc_PBKDF2**(byte * output, const byte * passwd, int pLen, const byte * salt, int sLen, int iterations, int kLen, int hashType)This function implements the Password Based Key Derivation Function 2 (PBKDF2), converting an input password with a concatenated salt into a more secure key, which it stores in output. It allows the user to select any of the supported HMAC hash functions, including: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512.
int	**wc_PKCS12_PBKDF**(byte * output, const byte * passwd, int passLen, const byte * salt, int saltLen, int iterations, int kLen, int hashType, int id)This function implements the Password Based Key Derivation Function (PBKDF) described in RFC 7292 Appendix B. This function converts an input password with a concatenated salt into a more secure key, which it stores in output. It allows the user to select any of the supported HMAC hash functions, including: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512.

	Name
int	**wc_PBKDF1_ex**(byte * key, int keyLen, byte * iv, int ivLen, const byte * passwd, int passwdLen, const byte * salt, int saltLen, int iterations, int hashType, void * heap)Extended version of PBKDF1 with heap hint.
int	**wc_PBKDF2_ex**(byte * output, const byte * passwd, int pLen, const byte * salt, int sLen, int iterations, int kLen, int hashType, void * heap, int devId)Extended version of PBKDF2 with heap hint and device ID.
int	**wc_PKCS12_PBKDF_ex**(byte * output, const byte * passwd, int passLen, const byte * salt, int saltLen, int iterations, int kLen, int hashType, int id, void * heap)Extended version of PKCS12_PBKDF with heap hint.
int	**wc_scrypt**(byte * output, const byte * passwd, int passLen, const byte * salt, int saltLen, int cost, int blockSize, int parallel, int dkLen)Implements scrypt key derivation function.
int	**wc_scrypt_ex**(byte * output, const byte * passwd, int passLen, const byte * salt, int saltLen, word32 iterations, int blockSize, int parallel, int dkLen)Extended scrypt with iteration count instead of cost.
int	**wc_PBKDF_max_iterations_set**(int iters)Set the current iteration limit for PBKDF.
int	**wc_PBKDF_max_iterations_get**(void)Get the current iteration limit for PBKDF.

C.45.2 Functions Documentation

C.45.2.1 function wc_PBKDF1

```
int wc_PBKDF1(
    byte * output,
    const byte * passwd,
    int pLen,
    const byte * salt,
    int sLen,
    int iterations,
    int kLen,
    int hashType
)
```

This function implements the Password Based Key Derivation Function 1 (PBKDF1), converting an input password with a concatenated salt into a more secure key, which it stores in output. It allows the user to select between SHA and MD5 as hash functions.

Parameters:

- **output** pointer to the buffer in which to store the generated key. Should be at least kLen long
- **passwd** pointer to the buffer containing the password to use for the key derivation
- **pLen** length of the password to use for key derivation

- **salt** pointer to the buffer containing the salt to use for key derivation
- **sLen** length of the salt
- **iterations** number of times to process the hash
- **kLen** desired length of the derived key. Should not be longer than the digest size of the hash chosen
- **hashType** the hashing algorithm to use. Valid choices are WC_MD5 and WC_SHA

See:

- wc_PBKDF2
- wc_PKCS12_PBKDF

Return:

- 0 Returned on successfully deriving a key from the input password
- BAD_FUNC_ARG Returned if there is an invalid hash type given (valid type are: MD5 and SHA), iterations is less than 1, or the key length (kLen) requested is greater than the hash length of the provided hash
- MEMORY_E Returned if there is an error allocating memory for a SHA or MD5 object

Example

```
int ret;
byte key[WC_MD5_DIGEST_SIZE];
byte pass[] = { }; // initialize with password
byte salt[] = { }; // initialize with salt

ret = wc_PBKDF1(key, pass, sizeof(pass), salt, sizeof(salt), 1000,
sizeof(key), WC_MD5);
if ( ret != 0 ) {
    // error deriving key from password
}
```

C.45.2.2 function wc_PBKDF2

```
int wc_PBKDF2(
    byte * output,
    const byte * passwd,
    int pLen,
    const byte * salt,
    int sLen,
    int iterations,
    int kLen,
    int hashType
)
```

This function implements the Password Based Key Derivation Function 2 (PBKDF2), converting an input password with a concatenated salt into a more secure key, which it stores in output. It allows the user to select any of the supported HMAC hash functions, including: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512.

Parameters:

- **output** pointer to the buffer in which to store the generated key. Should be kLen long
- **passwd** pointer to the buffer containing the password to use for the key derivation
- **pLen** length of the password to use for key derivation
- **salt** pointer to the buffer containing the salt to use for key derivation
- **sLen** length of the salt
- **iterations** number of times to process the hash

- **kLen** desired length of the derived key
- **hashType** the hashing algorithm to use. Valid choices are: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512

See:

- wc_PBKDF1
- wc_PKCS12_PBKDF

Return:

- 0 Returned on successfully deriving a key from the input password
- BAD_FUNC_ARG Returned if there is an invalid hash type given or iterations is less than 1
- MEMORY_E Returned if there is an allocating memory for the HMAC object

Example

```
int ret;
byte key[64];
byte pass[] = { }; // initialize with password
byte salt[] = { }; // initialize with salt

ret = wc_PBKDF2(key, pass, sizeof(pass), salt, sizeof(salt), 2048, sizeof(key),
WC_SHA512);
if ( ret != 0 ) {
    // error deriving key from password
}
```

C.45.2.3 function wc_PKCS12_PBKDF

```
int wc_PKCS12_PBKDF(
    byte * output,
    const byte * passwd,
    int passLen,
    const byte * salt,
    int saltLen,
    int iterations,
    int kLen,
    int hashType,
    int id
)
```

This function implements the Password Based Key Derivation Function (PBKDF) described in RFC 7292 Appendix B. This function converts an input password with a concatenated salt into a more secure key, which it stores in output. It allows the user to select any of the supported HMAC hash functions, including: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512.

Parameters:

- **output** pointer to the buffer in which to store the generated key. Should be kLen long
- **passwd** pointer to the buffer containing the password to use for the key derivation
- **passLen** length of the password to use for key derivation
- **salt** pointer to the buffer containing the salt to use for key derivation
- **saltLen** length of the salt
- **iterations** number of times to process the hash
- **kLen** desired length of the derived key
- **hashType** the hashing algorithm to use. Valid choices are: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512

- **id** this is a byte identifier indicating the purpose of key generation. It is used to diversify the key output, and should be assigned as follows: ID=1: pseudorandom bits are to be used as key material for performing encryption or decryption. ID=2: pseudorandom bits are to be used an IV (Initial Value) for encryption or decryption. ID=3: pseudorandom bits are to be used as an integrity key for MACing.

See:

- wc_PBKDF1
- wc_PBKDF2

Return:

- 0 Returned on successfully deriving a key from the input password
- BAD_FUNC_ARG Returned if there is an invalid hash type given, iterations is less than 1, or the key length (kLen) requested is greater than the hash length of the provided hash
- MEMORY_E Returned if there is an allocating memory
- MP_INIT_E may be returned if there is an error during key generation
- MP_READ_E may be returned if there is an error during key generation
- MP_CMP_E may be returned if there is an error during key generation
- MP_INVMOD_E may be returned if there is an error during key generation
- MP_EXPTMOD_E may be returned if there is an error during key generation
- MP_MOD_E may be returned if there is an error during key generation
- MP_MUL_E may be returned if there is an error during key generation
- MP_ADD_E may be returned if there is an error during key generation
- MP_MULMOD_E may be returned if there is an error during key generation
- MP_TO_E may be returned if there is an error during key generation
- MP_MEM may be returned if there is an error during key generation

Example

```
int ret;
byte key[64];
byte pass[] = { }; // initialize with password
byte salt[] = { }; // initialize with salt

ret = wc_PKCS12_PBKDF(key, pass, sizeof(pass), salt, sizeof(salt), 2048,
sizeof(key), WC_SHA512, 1);
if ( ret != 0 ) {
    // error deriving key from password
}
```

C.45.2.4 function wc_PBKDF1_ex

```
int wc_PBKDF1_ex(
    byte * key,
    int keyLen,
    byte * iv,
    int ivLen,
    const byte * passwd,
    int passwdLen,
    const byte * salt,
    int saltLen,
    int iterations,
    int hashType,
    void * heap
)
```

Extended version of PBKDF1 with heap hint.

Parameters:

- **key** Output key buffer
- **keyLen** Key length
- **iv** Output IV buffer
- **ivLen** IV length
- **passwd** Password buffer
- **passwdLen** Password length
- **salt** Salt buffer
- **saltLen** Salt length
- **iterations** Iteration count
- **hashType** Hash algorithm type
- **heap** Heap hint for memory allocation

See:

- wc_PBKDF1
- wc_PBKDF_max_iterations_set
- wc_PBKDF_max_iterations_get

Return:

- 0 on success
- BAD_FUNC_ARG on invalid arguments or iterations is greater than current_wc_pbkdf_max_iterations
- MEMORY_E on memory allocation error

Example

```
byte key[16], iv[16];
byte pass[] = "password";
byte salt[] = "salt";
int ret = wc_PBKDF1_ex(key, sizeof(key), iv, sizeof(iv),
    pass, sizeof(pass), salt, sizeof(salt), 1000, WC_SHA, NULL);
```

C.45.2.5 function wc_PBKDF2_ex

```
int wc_PBKDF2_ex(
    byte * output,
    const byte * passwd,
    int pLen,
    const byte * salt,
    int sLen,
    int iterations,
    int kLen,
    int hashType,
    void * heap,
    int devId
)
```

Extended version of PBKDF2 with heap hint and device ID.

Parameters:

- **output** Output key buffer
- **passwd** Password buffer
- **pLen** Password length
- **salt** Salt buffer
- **sLen** Salt length

- **iterations** Iteration count
- **kLen** Key length
- **hashType** Hash algorithm type
- **heap** Heap hint for memory allocation
- **devId** Device ID for hardware acceleration

See:

- wc_PBKDF2
- wc_PBKDF_max_iterations_set
- wc_PBKDF_max_iterations_get

Return:

- 0 on success
- BAD_FUNC_ARG on invalid arguments or iterations is greater than current_wc_pbkdf_max_iterations
- MEMORY_E on memory allocation error

Example

```
byte key[32];
byte pass[] = "password";
byte salt[] = "salt";
int ret = wc_PBKDF2_ex(key, pass, sizeof(pass), salt,
    sizeof(salt), 2048, sizeof(key), WC_SHA256, NULL,
    INVALID_DEVID);
```

C.45.2.6 function wc_PKCS12_PBKDF_ex

```
int wc_PKCS12_PBKDF_ex(
    byte * output,
    const byte * passwd,
    int passLen,
    const byte * salt,
    int saltLen,
    int iterations,
    int kLen,
    int hashType,
    int id,
    void * heap
)
```

Extended version of PKCS12_PBKDF with heap hint.

Parameters:

- **output** Output key buffer
- **passwd** Password buffer
- **passLen** Password length
- **salt** Salt buffer
- **saltLen** Salt length
- **iterations** Iteration count
- **kLen** Key length
- **hashType** Hash algorithm type
- **id** Purpose identifier (1=key, 2=IV, 3=MAC)
- **heap** Heap hint for memory allocation

See:

- wc_PKCS12_PBKDF

- wc_PBKDF_max_iterations_set
- wc_PBKDF_max_iterations_get

Return:

- 0 on success
- BAD_FUNC_ARG on invalid arguments or iterations is greater than current_wc_pbkdf_max_iterations
- MEMORY_E on memory allocation error

Example

```
byte key[32];
byte pass[] = "password";
byte salt[] = "salt";
int ret = wc_PKCS12_PBKDF_ex(key, pass, sizeof(pass), salt,
    sizeof(salt), 2048, sizeof(key), WC_SHA256, 1, NULL);
```

C.45.2.7 function wc_scrypt

```
int wc_scrypt(
    byte * output,
    const byte * passwd,
    int passLen,
    const byte * salt,
    int saltLen,
    int cost,
    int blockSize,
    int parallel,
    int dkLen
)
```

Implements scrypt key derivation function.

Parameters:

- **output** Output key buffer
- **passwd** Password buffer
- **passLen** Password length
- **salt** Salt buffer
- **saltLen** Salt length
- **cost** CPU/memory cost parameter (N)
- **blockSize** Block size parameter (r)
- **parallel** Parallelization parameter (p)
- **dkLen** Derived key length

See: wc_scrypt_ex

Return:

- 0 on success
- BAD_FUNC_ARG on invalid arguments
- MEMORY_E on memory allocation error

Example

```
byte key[32];
byte pass[] = "password";
byte salt[] = "salt";
int ret = wc_scrypt(key, pass, sizeof(pass), salt,
    sizeof(salt), 16384, 8, 1, sizeof(key));
```

C.45.2.8 function wc_scrypt_ex

```
int wc_scrypt_ex(
    byte * output,
    const byte * passwd,
    int passLen,
    const byte * salt,
    int saltLen,
    word32 iterations,
    int blockSize,
    int parallel,
    int dkLen
)
```

Extended scrypt with iteration count instead of cost.

Parameters:

- **output** Output key buffer
- **passwd** Password buffer
- **passLen** Password length
- **salt** Salt buffer
- **saltLen** Salt length
- **iterations** Iteration count
- **blockSize** Block size parameter (r)
- **parallel** Parallelization parameter (p)
- **dkLen** Derived key length

See: wc_scrypt

Return:

- 0 on success
- BAD_FUNC_ARG on invalid arguments
- MEMORY_E on memory allocation error

Example

```
byte key[32];
byte pass[] = "password";
byte salt[] = "salt";
int ret = wc_scrypt_ex(key, pass, sizeof(pass), salt,
    sizeof(salt), 16384, 8, 1, sizeof(key));
```

C.45.2.9 function wc_PBKDF_max_iterations_set

```
int wc_PBKDF_max_iterations_set(
    int iters
)
```

Set the current iteration limit for PBKDF.

Parameters:

- **iters** The new iteration limit.

See: wc_scrypt

Return:

- Previous iteration limit on success
- BAD_FUNC_ARG on invalid arguments

By default, the iteration limit is set to WC_PBKDF_DEFAULT_MAX_ITERATIONS, which can be overridden at build time. This function allows runtime override of the limit.

Note that [wc_PBKDF_max_iterations_set()](#function-wc_pbkdf_max_iterations_set) has no provisions for thread synchronization. Users should arrange to call it at startup or idle times, when there are no other PBKDF calls in progress.

Example

```
int prev_iter_limit = wc_PBKDF_max_iterations_set(100000000);
```

C.45.2.10 function wc_PBKDF_max_iterations_get

```
int wc_PBKDF_max_iterations_get(
    void
)
```

Get the current iteration limit for PBKDF.

See: wc_scrypt

Return: Current iteration limit

Example

```
int cur_iter_limit = wc_PBKDF_max_iterations_get();
```

C.45.3 Source code

```
int wc_PBKDF1(byte* output, const byte* passwd, int pLen,
                      const byte* salt, int sLen, int iterations, int kLen,
                      int hashType);

int wc_PBKDF2(byte* output, const byte* passwd, int pLen,
                      const byte* salt, int sLen, int iterations, int kLen,
                      int hashType);

int wc_PKCS12_PBKDF(byte* output, const byte* passwd, int passLen,
                            const byte* salt, int saltLen, int iterations,
                            int kLen, int hashType, int id);

int wc_PBKDF1_ex(byte* key, int keyLen, byte* iv, int ivLen,
    const byte* passwd, int passwdLen, const byte* salt, int saltLen,
    int iterations, int hashType, void* heap);

int wc_PBKDF2_ex(byte* output, const byte* passwd, int pLen,
    const byte* salt, int sLen, int iterations, int kLen,
    int hashType, void* heap, int devId);

int wc_PKCS12_PBKDF_ex(byte* output, const byte* passwd,int passLen,
    const byte* salt, int saltLen, int iterations, int kLen,
    int hashType, int id, void* heap);

int wc_scrypt(byte* output, const byte* passwd, int passLen,
    const byte* salt, int saltLen, int cost, int blockSize,
    int parallel, int dkLen);
```

```
int wc_scrypt_ex(byte* output, const byte* passwd, int passLen,
    const byte* salt, int saltLen, word32 iterations, int blockSize,
    int parallel, int dkLen);

int wc_PBKDF_max_iterations_set(int iters);

int wc_PBKDF_max_iterations_get(void);
```

C.46 quic.h

C.46.1 Functions

	Name
int	**wolfSSL_CTX_set_quic_method**(WOLFSSL_CTX * ctx, const WOLFSSL_QUIC_METHOD * quic_method)Activate QUIC protocol for a WOLFSSL_CTX and all derived WOLFSSL instances by providing the four callbacks required. The CTX needs to be a TLSv1.3 one.
int	**wolfSSL_set_quic_method**(WOLFSSL * ssl, const WOLFSSL_QUIC_METHOD * quic_method)Activate QUIC protocol for a WOLFSSL instance by providing the four callbacks required. The WOLFSSL needs to be a TLSv1.3 one.
int	**wolfSSL_is_quic**(WOLFSSL * ssl)Check if QUIC has been activated in a WOLFSSL instance.
WOLFSSL_ENCRYPTION_LEVEL	**wolfSSL_quic_read_level**(const WOLFSSL * ssl)Determine the encryption level for reads currently in use. Meaningful only when the WOLFSSL instance is using QUIC.
WOLFSSL_ENCRYPTION_LEVEL	**wolfSSL_quic_write_level**(const WOLFSSL * ssl)Determine the encryption level for writes currently in use. Meaningful only when the WOLFSSL instance is using QUIC.
void	**wolfSSL_set_quic_use_legacy_codepoint**(WOLFSSL * ssl, int use_legacy)Configure which QUIC version shall be used. Without calling this, the WOLFSSL will offer both (draft_27 and v1) to a server, resp. accept both from a client and negotiate the most recent one.
void	**wolfSSL_set_quic_transport_version**(WOLFSSL * ssl, int version)Configure which QUIC version shall be used. An unknown version is ignored with a warning.
int	**wolfSSL_get_quic_transport_version**(const WOLFSSL * ssl)Get the configured QUIC version.
int	**wolfSSL_set_quic_transport_params**(WOLFSSL * ssl, const uint8_t * params, size_t params_len)Set the QUIC transport parameters to use.

	Name
int	**wolfSSL_get_peer_quic_transport_version**(const WOLFSSL * ssl)Get the negotiated QUIC transport version. This will only give meaningful results when called after the respective TLS extensions have been seen from the peer.
void	**wolfSSL_get_peer_quic_transport_params**(const WOLFSSL * ssl, const uint8_t ** out_params, size_t * out_params_len)Get the negotiated QUIC transport parameters. This will only give meaningful results when called after the respective TLS extensions have been seen from the peer.
void	**wolfSSL_set_quic_early_data_enabled**(WOLFSSL * ssl, int enabled)Configure if Early Data is enabled. Intended for servers to signal this to clients.
size_t	**wolfSSL_quic_max_handshake_flight_len**(const WOLFSSL * ssl, WOLFSSL_ENCRYPTION_LEVEL level)Get advice on the amount of data that shall be “in flight”, e.g. unacknowledged at the given encryption level. This is the amount of data the WOLFSSL instance is prepared to buffer.
int	**wolfSSL_provide_quic_data**(WOLFSSL * ssl, WOLFSSL_ENCRYPTION_LEVEL level, const uint8_t * data, size_t len)Pass decrypted CRYPTO data to the WOLFSSL instance for further processing. The encryption level between calls is only every allowed to increase and it is also checked that data records are complete before a change in encryption level is accepted.
WOLFSSL_API int	**wolfSSL_process_quic_post_handshake**(WOLFSSL * ssl)Process any CRYPTO records that have been provided after the handshake has completed. Will fail if called before that.
int	**wolfSSL_quic_read_write**(WOLFSSL * ssl)Process any CRYPTO records that have been provided during or after the handshake. Will progress the handshake if not already complete and otherwise work like wolfSSL_process_quic_post_handshake().
int	**wolfSSL_quic_do_handshake**(WOLFSSL * ssl)Perform the QUIC handshake. This function processes CRYPTO data that has been provided via wolfSSL_provide_quic_data() and advances the handshake state. It should be called repeatedly until the handshake is complete.
const WOLFSSL_EVP_CIPHER *	**wolfSSL_quic_get_aead**(WOLFSSL * ssl)Get the AEAD cipher negotiated in the TLS handshake.

	Name
int	**wolfSSL_quic_aead_is_gcm**(const WOLFSSL_EVP_CIPHER * aead_cipher)Check if the AEAD cipher is GCM.
int	**wolfSSL_quic_aead_is_ccm**(const WOLFSSL_EVP_CIPHER * aead_cipher)Check if the AEAD cipher is CCM.
int	**wolfSSL_quic_aead_is_chacha20**(const WOLFSSL_EVP_CIPHER * aead_cipher)Check if the AEAD cipher is CHACHA20.
WOLFSSL_API size_t	**wolfSSL_quic_get_aead_tag_len**(const WOLFSSL_EVP_CIPHER * aead_cipher)Determine the tag length for the AEAD cipher.
WOLFSSL_API const WOLFSSL_EVP_MD *	**wolfSSL_quic_get_md**(WOLFSSL * ssl)Determine the message digest negotiated in the TLS handshake.
const WOLFSSL_EVP_CIPHER *	**wolfSSL_quic_get_hp**(WOLFSSL * ssl)Determine the header protection cipher negotiated in the TLS handshake.
WOLFSSL_EVP_CIPHER_CTX *	**wolfSSL_quic_crypt_new**(const WOLFSSL_EVP_CIPHER * cipher, const uint8_t * key, const uint8_t * iv, int encrypt)Create a cipher context for en-/decryption.
int	**wolfSSL_quic_aead_encrypt**(uint8_t * dest, WOLFSSL_EVP_CIPHER_CTX * aead_ctx, const uint8_t * plain, size_t plainlen, const uint8_t * iv, const uint8_t * aad, size_t aadlen)Encrypt the plain text in the given context.
int	**wolfSSL_quic_aead_decrypt**(uint8_t * dest, WOLFSSL_EVP_CIPHER_CTX * ctx, const uint8_t * enc, size_t enclen, const uint8_t * iv, const uint8_t * aad, size_t aadlen)Decrypt the cipher text in the given context.
int	**wolfSSL_quic_hkdf_extract**(uint8_t * dest, const WOLFSSL_EVP_MD * md, const uint8_t * secret, size_t secretlen, const uint8_t * salt, size_t saltlen)Extract a pseudo random key.
int	**wolfSSL_quic_hkdf_expand**(uint8_t * dest, size_t destlen, const WOLFSSL_EVP_MD * md, const uint8_t * secret, size_t secretlen, const uint8_t * info, size_t infolen)Expand a pseudo random key into a new key.
int	**wolfSSL_quic_hkdf**(uint8_t * dest, size_t destlen, const WOLFSSL_EVP_MD * md, const uint8_t * secret, size_t secretlen, const uint8_t * salt, size_t saltlen, const uint8_t * info, size_t infolen)Expand and Extract a pseudo random key.

C.46.2 Attributes

	Name
int()(*WOLFSSL* ssl, WOLFSSL_ENCRYPTION_LEVEL level, const uint8_t *read_secret, const uint8_t* write_secret, size_t secret_len)	**set_encryption_secrets** Callback invoked when secrets are generated during a handshake. Since QUIC protocol handlers perform the en-/decryption of packets, they need the negotiated secrets for the levels early_data/handshake/application.
int()(*WOLFSSL* ssl, WOLFSSL_ENCRYPTION_LEVEL level, const uint8_t *data, size_t len)	**add_handshake_data** Callback invoked for forwarding handshake CRYPTO data to peer. The data forwarded this way is not encrypted. It is the job of the QUIC protocol implementation to do this. Which secrets are to be used is determined by the encryption level specified.
int()(*WOLFSSL* ssl)	**flush_flight** Callback invoked for advisory flushing of the data to send.
int()(*WOLFSSL* ssl, WOLFSSL_ENCRYPTION_LEVEL level, uint8_t alert)	**send_alert** Callback invoked when an SSL alert happened during processing.

C.46.3 Functions Documentation

C.46.3.1 function wolfSSL_CTX_set_quic_method

```
int wolfSSL_CTX_set_quic_method(
    WOLFSSL_CTX * ctx,
    const WOLFSSL_QUIC_METHOD * quic_method
)
```

Activate QUIC protocol for a WOLFSSL_CTX and all derived WOLFSSL instances by providing the four callbacks required. The CTX needs to be a TLSv1.3 one.

Parameters:

- **ctx** - a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **quic_method** - the callback structure

See:

- wolfSSL_is_quic
- wolfSSL_set_quic_method

Return:

- WOLFSSL_SUCCESS If successful.
- WOLFSSL_FAILURE If not TLSv1.3 or a callback is missing.

The passed quic_method needs to have a lifetime outlasting the SSL instances. It is not copied. All callbacks need to be provided.

C.46.3.2 function wolfSSL_set_quic_method

```
int wolfSSL_set_quic_method(
    WOLFSSL * ssl,
    const WOLFSSL_QUIC_METHOD * quic_method
)
```

Activate QUIC protocol for a WOLFSSL instance by providing the four callbacks required. The WOLFSSL needs to be a TLSv1.3 one.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **quic_method** - the callback structure

See:

- wolfSSL_is_quic
- wolfSSL_CTX_set_quic_method

Return:

- WOLFSSL_SUCCESS If successful.
- WOLFSSL_FAILURE If not TLSv1.3 or a callback is missing.

The passed quic_method needs to have a lifetime outlasting the SSL instance. It is not copied. All callbacks need to be provided.

C.46.3.3 function wolfSSL_is_quic

```
int wolfSSL_is_quic(
    WOLFSSL * ssl
)
```

Check if QUIC has been activated in a WOLFSSL instance.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_CTX_quic_method
- wolfSSL_CTX_set_quic_method

Return:

- 1 if WOLFSSL is using QUIC.
- 0 otherwise.

C.46.3.4 function wolfSSL_quic_read_level

```
WOLFSSL_ENCRYPTION_LEVEL wolfSSL_quic_read_level(
    const WOLFSSL * ssl
)
```

Determine the encryption level for reads currently in use. Meaningful only when the WOLFSSL instance is using QUIC.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_quic_write_level

Return: encryption level.

Note that the effective level is always a parameter when passing data back and forth. Data from a peer might arrive at other levels than reported via this function.

C.46.3.5 function wolfSSL_quic_write_level

```
WOLFSSL_ENCRYPTION_LEVEL wolfSSL_quic_write_level(
    const WOLFSSL * ssl
)
```

Determine the encryption level for writes currently in use. Meaningful only when the WOLFSSL instance is using QUIC.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_quic_read_level

Return: encryption level.

Note that the effective level is always a parameter when passing data back and forth. Data from a peer might arrive at other levels than reported via this function.

C.46.3.6 function wolfSSL_set_quic_use_legacy_codepoint

```
void wolfSSL_set_quic_use_legacy_codepoint(
    WOLFSSL * ssl,
    int use_legacy
)
```

Configure which QUIC version shall be used. Without calling this, the WOLFSSL will offer both (draft-27 and v1) to a server, resp. accept both from a client and negotiate the most recent one.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **use_legacy** - true if draft-27 shall be used, 0 if only QUICv1 is used.

See: wolfSSL_set_quic_transport_version

C.46.3.7 function wolfSSL_set_quic_transport_version

```
void wolfSSL_set_quic_transport_version(
    WOLFSSL * ssl,
    int version
)
```

Configure which QUIC version shall be used. An unknown version is ignored with a warning.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **version** - the TLS Extension defined for the QUIC version.

See: wolfSSL_set_quic_use_legacy_codepoint

C.46.3.8 function wolfSSL_get_quic_transport_version

```
int wolfSSL_get_quic_transport_version(
    const WOLFSSL * ssl
)
```

Get the configured QUIC version.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_set_quic_use_legacy_codepoint
- wolfSSL_set_quic_transport_version

Return: TLS Extension of configured version.

C.46.3.9 function wolfSSL_set_quic_transport_params

```
int wolfSSL_set_quic_transport_params(
    WOLFSSL * ssl,
    const uint8_t * params,
    size_t params_len
)
```

Set the QUIC transport parameters to use.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **params** - the parameter bytes to use ·param params_len - the length of the parameters

See:

- wolfSSL_set_quic_use_legacy_codepoint
- wolfSSL_set_quic_transport_version

Return:

- WOLFSSL_SUCCESS If successful.
- WOLFSSL_FAILURE If the parameters could not be stored.

C.46.3.10 function wolfSSL_get_peer_quic_transport_version

```
int wolfSSL_get_peer_quic_transport_version(
    const WOLFSSL * ssl
)
```

Get the negotiated QUIC transport version. This will only give meaningful results when called after the respective TLS extensions have been seen from the peer.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_set_quic_use_legacy_codepoint
- wolfSSL_set_quic_transport_version

Return: the negotiated version or -1.

C.46.3.11 function wolfSSL_get_peer_quic_transport_params

```
void wolfSSL_get_peer_quic_transport_params(
    const WOLFSSL * ssl,
    const uint8_t ** out_params,
    size_t * out_params_len
)
```

Get the negotiated QUIC transport parameters. This will only give meaningful results when called after the respective TLS extensions have been seen from the peer.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **out_params** - the parameters sent be the peer, set to NULL if not available.
- **out_params_len** - the length of the parameters sent be the peer, set to 0 if not available

See: wolfSSL_get_peer_quic_transport_version

C.46.3.12 function wolfSSL_set_quic_early_data_enabled

```
void wolfSSL_set_quic_early_data_enabled(
    WOLFSSL * ssl,
    int enabled
)
```

Configure if Early Data is enabled. Intended for servers to signal this to clients.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **enabled** - != 0 iff early data is enabled

C.46.3.13 function wolfSSL_quic_max_handshake_flight_len

```
size_t wolfSSL_quic_max_handshake_flight_len(
    const WOLFSSL * ssl,
    WOLFSSL_ENCRYPTION_LEVEL level
)
```

Get advice on the amount of data that shall be “in flight”, e.g. unacknowledged at the given encryption level. This is the amount of data the WOLFSSL instance is prepared to buffer.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **level** - the encryption level to inquire about

Return: the recommend max data in flight

C.46.3.14 function wolfSSL_provide_quic_data

```
int wolfSSL_provide_quic_data(
    WOLFSSL * ssl,
    WOLFSSL_ENCRYPTION_LEVEL level,
    const uint8_t * data,
    size_t len
)
```

Pass decrypted CRYPTO data to the WOLFSSL instance for further processing. The encryption level between calls is only every allowed to increase and it is also checked that data records are complete before a change in encryption level is accepted.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **level** - the level the data was encrypted at
- **data** - the data itself
- **len** - the length of the data

See:

- wolfSSL_process_quic_post_handshake
- wolfSSL_quic_read_write
- wolfSSL_accept
- wolfSSL_connect

Return:

- WOLFSSL_SUCCESS If successful.
- WOLFSSL_FAILURE On any error, the only other value returned, as the quictls SSL_provide_quic_data() defines. Every failure path leaves a reason that wolfSSL_get_error() returns. Not recoverable, the connection has to be failed.

C.46.3.15 function wolfSSL_process_quic_post_handshake

```
WOLFSSL_API int wolfSSL_process_quic_post_handshake(
    WOLFSSL * ssl
)
```

Process any CRYPTO records that have been provided after the handshake has completed. Will fail if called before that.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_provide_quic_data
- wolfSSL_quic_read_write
- wolfSSL_accept
- wolfSSL_connect

Return:

- WOLFSSL_SUCCESS If successful.
- WOLFSSL_FAILURE On any error, the only other value returned, as the quictls SSL_process_quic_post_handshake() defines. Every failure path leaves a reason that wolfSSL_get_error() returns.

C.46.3.16 function wolfSSL_quic_read_write

```
int wolfSSL_quic_read_write(
    WOLFSSL * ssl
)
```

Process any CRYPTO records that have been provided during or after the handshake. Will progress the handshake if not already complete and otherwise work like wolfSSL_process_quic_post_handshake().

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_provide_quic_data
- wolfSSL_quic_read_write
- wolfSSL_accept
- wolfSSL_connect

Return:

- WOLFSSL_SUCCESS If successful.

- WOLFSSL_FAILURE If not a QUIC WOLFSSL or the post-handshake processing failed; use wolfSSL_get_error() for the reason.
- Other values from wolfSSL_quic_do_handshake() while the handshake is in progress; use wolfSSL_get_error() to tell WANT_READ from a failure.

C.46.3.17 function wolfSSL_quic_do_handshake

```
int wolfSSL_quic_do_handshake(
    WOLFSSL * ssl
)
```

Perform the QUIC handshake. This function processes CRYPTO data that has been provided via wolfSSL_provide_quic_data() and advances the handshake state. It should be called repeatedly until the handshake is complete.

Parameters:

- **ssl** pointer to a WOLFSSL structure created using wolfSSL_new()

See:

- wolfSSL_provide_quic_data
- wolfSSL_quic_read_write
- wolfSSL_is_init_finished

Return:

- WOLFSSL_SUCCESS If handshake completed successfully
- WOLFSSL_FATAL_ERROR If a fatal error occurred
- Other values indicating handshake is in progress

Example

```
WOLFSSL* ssl;
// initialize ssl with QUIC method

while (!wolfSSL_is_init_finished(ssl)) {
    int ret = wolfSSL_quic_do_handshake(ssl);
    if (ret == WOLFSSL_FATAL_ERROR) {
        // handle error
        break;
    }
    // provide more CRYPTO data if available
}
```

C.46.3.18 function wolfSSL_quic_get_aead

```
const WOLFSSL_EVP_CIPHER * wolfSSL_quic_get_aead(
    WOLFSSL * ssl
)
```

Get the AEAD cipher negotiated in the TLS handshake.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_quic_aead_is_gcm
- wolfSSL_quic_aead_is_ccm

- wolfSSL_quic_aead_is_chacha20
- wolfSSL_quic_get_aead_tag_len
- wolfSSL_quic_get_md
- wolfSSL_quic_get_hp
- wolfSSL_quic_crypt_new
- wolfSSL_quic_aead_encrypt
- wolfSSL_quic_aead_decrypt

Return: negotiated cipher or NULL if not determined.

C.46.3.19 function wolfSSL_quic_aead_is_gcm

```
int wolfSSL_quic_aead_is_gcm(
    const WOLFSSL_EVP_CIPHER * aead_cipher
)
```

Check if the AEAD cipher is GCM.

Parameters:

- **cipher** - the cipher

See:

- wolfSSL_quic_get_aead
- wolfSSL_quic_aead_is_ccm
- wolfSSL_quic_aead_is_chacha20
- wolfSSL_quic_get_aead_tag_len
- wolfSSL_quic_get_md
- wolfSSL_quic_get_hp
- wolfSSL_quic_crypt_new
- wolfSSL_quic_aead_encrypt
- wolfSSL_quic_aead_decrypt

Return: != 0 iff the AEAD cipher is GCM.

C.46.3.20 function wolfSSL_quic_aead_is_ccm

```
int wolfSSL_quic_aead_is_ccm(
    const WOLFSSL_EVP_CIPHER * aead_cipher
)
```

Check if the AEAD cipher is CCM.

Parameters:

- **cipher** - the cipher

See:

- wolfSSL_quic_get_aead
- wolfSSL_quic_aead_is_gcm
- wolfSSL_quic_aead_is_chacha20
- wolfSSL_quic_get_aead_tag_len
- wolfSSL_quic_get_md
- wolfSSL_quic_get_hp
- wolfSSL_quic_crypt_new
- wolfSSL_quic_aead_encrypt
- wolfSSL_quic_aead_decrypt

Return: != 0 iff the AEAD cipher is CCM.

C.46.3.21 function wolfSSL_quic_aead_is_chacha20

```
int wolfSSL_quic_aead_is_chacha20(
    const WOLFSSL_EVP_CIPHER * aead_cipher
)
```

Check if the AEAD cipher is CHACHA20.

Parameters:

- **cipher** - the cipher

See:

- wolfSSL_quic_get_aead
- wolfSSL_quic_aead_is_ccm
- wolfSSL_quic_aead_is_gcm
- wolfSSL_quic_get_aead_tag_len
- wolfSSL_quic_get_md
- wolfSSL_quic_get_hp
- wolfSSL_quic_crypt_new
- wolfSSL_quic_aead_encrypt
- wolfSSL_quic_aead_decrypt

Return: != 0 iff the AEAD cipher is CHACHA20.

C.46.3.22 function wolfSSL_quic_get_aead_tag_len

```
WOLFSSL_API size_t wolfSSL_quic_get_aead_tag_len(
    const WOLFSSL_EVP_CIPHER * aead_cipher
)
```

Determine the tag length for the AEAD cipher.

Parameters:

- **cipher** - the cipher

See: wolfSSL_quic_get_aead

Return: tag length of AEAD cipher.

C.46.3.23 function wolfSSL_quic_get_md

```
WOLFSSL_API const WOLFSSL_EVP_MD * wolfSSL_quic_get_md(
    WOLFSSL * ssl
)
```

Determine the message digest negotiated in the TLS handshake.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_quic_get_aead
- wolfSSL_quic_get_hp

Return: the message digest negotiated in the TLS handshake

C.46.3.24 function wolfSSL_quic_get_hp

```
const WOLFSSL_EVP_CIPHER * wolfSSL_quic_get_hp(
    WOLFSSL * ssl
)
```

Determine the header protection cipher negotiated in the TLS handshake.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_quic_get_aead
- wolfSSL_quic_get_md

Return: the header protection cipher negotiated in the TLS handshake

C.46.3.25 function wolfSSL_quic_crypt_new

```
WOLFSSL_EVP_CIPHER_CTX * wolfSSL_quic_crypt_new(
    const WOLFSSL_EVP_CIPHER * cipher,
    const uint8_t * key,
    const uint8_t * iv,
    int encrypt
)
```

Create a cipher context for en-/decryption.

Parameters:

- **cipher** - the cipher to use in the context.
- **key** - the key to use in the context.
- **iv** - the iv to use in the context.
- **encrypt** - != 0 if for encryption, otherwise decryption

See:

- wolfSSL_quic_get_aead
- wolfSSL_quic_get_hp
- wolfSSL_quic_aead_encrypt
- wolfSSL_quic_aead_decrypt

Return: the created context or NULL in case of errors.

C.46.3.26 function wolfSSL_quic_aead_encrypt

```
int wolfSSL_quic_aead_encrypt(
    uint8_t * dest,
    WOLFSSL_EVP_CIPHER_CTX * aead_ctx,
    const uint8_t * plain,
    size_t plainlen,
    const uint8_t * iv,
    const uint8_t * aad,
    size_t aadlen
)
```

Encrypt the plain text in the given context.

Parameters:

- **dest** - destination where encrypted data is to be written
- **aead_ctx** - the cipher context to use
- **plain** - the plain data to encrypt
- **plainlen** - the length of the plain data
- **iv** - the iv to use
- **aad** - the add to use
- **aadlen** - the length of the aad

See:

- wolfSSL_quic_get_aead
- wolfSSL_quic_get_hp
- wolfSSL_quic_crypt_new
- wolfSSL_quic_aead_decrypt

Return:

- WOLFSSL_SUCCESS If successful.
- WOLFSSL_FAILURE If the encryption failed.

C.46.3.27 function wolfSSL_quic_aead_decrypt

```
int wolfSSL_quic_aead_decrypt(
    uint8_t * dest,
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    const uint8_t * enc,
    size_t enclen,
    const uint8_t * iv,
    const uint8_t * aad,
    size_t aadlen
)
```

Decrypt the cipher text in the given context.

Parameters:

- **dest** - destination where plain text is to be written
- **ctx** - the cipher context to use
- **enc** - the encrypted data to decrypt
- **envlen** - the length of the encrypted data
- **iv** - the iv to use
- **aad** - the add to use
- **aadlen** - the length of the aad

See:

- wolfSSL_quic_get_aead
- wolfSSL_quic_get_hp
- wolfSSL_quic_crypt_new
- wolfSSL_quic_aead_encrypt

Return:

- WOLFSSL_SUCCESS If successful.
- WOLFSSL_FAILURE If the decryption or the tag check failed.

C.46.3.28 function wolfSSL_quic_hkdf_extract

```
int wolfSSL_quic_hkdf_extract(
    uint8_t * dest,
```

```
    const WOLFSSL_EVP_MD * md,
    const uint8_t * secret,
    size_t secretlen,
    const uint8_t * salt,
    size_t saltlen
)
```

Extract a pseudo random key.

Parameters:

- **dest** - destination where key is to be written
- **md** - message digest to use
- **secret** - the secret to use
- **secretlen** - the length of the secret
- **salt** - the salt to use
- **saltlen** - the length of the salt

See:

- wolfSSL_quic_hkdf_expand
- wolfSSL_quic_hkdf

Return:

- WOLFSSL_SUCCESS If successful.
- WOLFSSL_FAILURE If the extract failed.

C.46.3.29 function wolfSSL_quic_hkdf_expand

```
int wolfSSL_quic_hkdf_expand(
    uint8_t * dest,
    size_t destlen,
    const WOLFSSL_EVP_MD * md,
    const uint8_t * secret,
    size_t secretlen,
    const uint8_t * info,
    size_t infolen
)
```

Expand a pseudo random key into a new key.

Parameters:

- **dest** - destination where key is to be written
- **destlen** - length of the key to expand
- **md** - message digest to use
- **secret** - the secret to use
- **secretlen** - the length of the secret
- **info** - the info to use
- **infolen** - the length of the info

See:

- wolfSSL_quic_hkdf_extract
- wolfSSL_quic_hkdf

Return:

- WOLFSSL_SUCCESS If successful.
- WOLFSSL_FAILURE If the expand failed.

C.46.3.30 function wolfSSL_quic_hkdf

```
int wolfSSL_quic_hkdf(
    uint8_t * dest,
    size_t destlen,
    const WOLFSSL_EVP_MD * md,
    const uint8_t * secret,
    size_t secretlen,
    const uint8_t * salt,
    size_t saltlen,
    const uint8_t * info,
    size_t infolen
)
```

Expand and Extract a pseudo random key.

Parameters:

- **dest** - destination where key is to be written
- **destlen** - length of the key
- **md** - message digest to use
- **secret** - the secret to use
- **secretlen** - the length of the secret
- **salt** - the salt to use
- **saltlen** - the length of the salt
- **info** - the info to use
- **infolen** - the length of the info

See:

- wolfSSL_quic_hkdf_extract
- wolfSSL_quic_hkdf_expand

Return:

- WOLFSSL_SUCCESS If successful.
- WOLFSSL_FAILURE If the extract or the expand failed.

C.46.4 Attributes Documentation

C.46.4.1 variable set_encryption_secrets

```
int(*)(WOLFSSL *ssl, WOLFSSL_ENCRYPTION_LEVEL level, const uint8_t
 ↪  *read_secret, const uint8_t *write_secret, size_t secret_len)
 ↪  set_encryption_secrets;
```

Callback invoked when secrets are generated during a handshake. Since QUIC protocol handlers perform the en-/decryption of packets, they need the negotiated secrets for the levels early_data/handshake/application.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **level** - the encryption level the secrets are for
- **read_secret** - the secret used in decryption at the given level, may be NULL.
- **write_secret** - the secret used in encryption at the given level, may be NULL.
- **secret_len** - the length of the secret

See: wolfSSL_set_quic_method

Return: 1 on success, 0 on failure.

The callback will be invoked several times during a handshake. Either both or only the read or write secret might be provided. This does not mean the given encryption level is already in effect.

C.46.4.2 variable add_handshake_data

```
int(*)(WOLFSSL *ssl, WOLFSSL_ENCRYPTION_LEVEL level, const uint8_t *data,
 ↪  size_t len) add_handshake_data;
```

Callback invoked for forwarding handshake CRYPTO data to peer. The data forwarded this way is not encrypted. It is the job of the QUIC protocol implementation to do this. Which secrets are to be used is determined by the encryption level specified.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **level** - the encryption level to use for encrypting the data
- **data** - the data itself
- **len** - the length of the data

See: wolfSSL_set_quic_method

Return: 1 on success, 0 on failure.

This callback may be invoked several times during handshake or post handshake processing. The data may cover a complete CRYPTO record, but may also be partial. However, the callback will have received all records data before using another encryption level.

C.46.4.3 variable flush_flight

```
int(*)(WOLFSSL *ssl) flush_flight;
```

Callback invoked for advisory flushing of the data to send.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_set_quic_method

Return: 1 on success, 0 on failure.

C.46.4.4 variable send_alert

```
int(*)(WOLFSSL *ssl, WOLFSSL_ENCRYPTION_LEVEL level, uint8_t alert) send_alert;
```

Callback invoked when an SSL alert happened during processing.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **level** - the encryption level in effect when the alert happened
- **alert** - the error

See: wolfSSL_set_quic_method

Return: 1 on success, 0 on failure.

C.46.5 Source code

```
int (*set_encryption_secrets)(WOLFSSL *ssl, WOLFSSL_ENCRYPTION_LEVEL level,
                              const uint8_t *read_secret,
```

```
                                const uint8_t *write_secret, size_t secret_len);

int (*add_handshake_data)(WOLFSSL *ssl, WOLFSSL_ENCRYPTION_LEVEL level,
                          const uint8_t *data, size_t len);

int (*flush_flight)(WOLFSSL *ssl);

int (*send_alert)(WOLFSSL *ssl, WOLFSSL_ENCRYPTION_LEVEL level, uint8_t alert);

int wolfSSL_CTX_set_quic_method(WOLFSSL_CTX *ctx, const WOLFSSL_QUIC_METHOD
 ↪  *quic_method);

int wolfSSL_set_quic_method(WOLFSSL *ssl, const WOLFSSL_QUIC_METHOD
 ↪  *quic_method);

int wolfSSL_is_quic(WOLFSSL *ssl);

WOLFSSL_ENCRYPTION_LEVEL wolfSSL_quic_read_level(const WOLFSSL *ssl);

WOLFSSL_ENCRYPTION_LEVEL wolfSSL_quic_write_level(const WOLFSSL *ssl);


void wolfSSL_set_quic_use_legacy_codepoint(WOLFSSL *ssl, int use_legacy);

void wolfSSL_set_quic_transport_version(WOLFSSL *ssl, int version);

int wolfSSL_get_quic_transport_version(const WOLFSSL *ssl);

int wolfSSL_set_quic_transport_params(WOLFSSL *ssl, const uint8_t *params,
 ↪  size_t params_len);

int wolfSSL_get_peer_quic_transport_version(const WOLFSSL *ssl);

void wolfSSL_get_peer_quic_transport_params(const WOLFSSL *ssl, const uint8_t
 ↪  **out_params, size_t *out_params_len);


void wolfSSL_set_quic_early_data_enabled(WOLFSSL *ssl, int enabled);

size_t wolfSSL_quic_max_handshake_flight_len(const WOLFSSL *ssl,
 ↪  WOLFSSL_ENCRYPTION_LEVEL level);


int wolfSSL_provide_quic_data(WOLFSSL *ssl, WOLFSSL_ENCRYPTION_LEVEL level,
 ↪  const uint8_t *data, size_t len);

WOLFSSL_API int wolfSSL_process_quic_post_handshake(WOLFSSL *ssl);

int wolfSSL_quic_read_write(WOLFSSL *ssl);

int wolfSSL_quic_do_handshake(WOLFSSL* ssl);

const WOLFSSL_EVP_CIPHER *wolfSSL_quic_get_aead(WOLFSSL *ssl);
```

```
int wolfSSL_quic_aead_is_gcm(const WOLFSSL_EVP_CIPHER *aead_cipher);

int wolfSSL_quic_aead_is_ccm(const WOLFSSL_EVP_CIPHER *aead_cipher);

int wolfSSL_quic_aead_is_chacha20(const WOLFSSL_EVP_CIPHER *aead_cipher);

WOLFSSL_API size_t wolfSSL_quic_get_aead_tag_len(const WOLFSSL_EVP_CIPHER
 ↪  *aead_cipher);

WOLFSSL_API const WOLFSSL_EVP_MD *wolfSSL_quic_get_md(WOLFSSL *ssl);

const WOLFSSL_EVP_CIPHER *wolfSSL_quic_get_hp(WOLFSSL *ssl);

WOLFSSL_EVP_CIPHER_CTX *wolfSSL_quic_crypt_new(const WOLFSSL_EVP_CIPHER
 ↪  *cipher,
                                               const uint8_t *key, const uint8_t
                                                 ↪  *iv, int encrypt);

int wolfSSL_quic_aead_encrypt(uint8_t *dest, WOLFSSL_EVP_CIPHER_CTX *aead_ctx,
                              const uint8_t *plain, size_t plainlen,
                              const uint8_t *iv, const uint8_t *aad, size_t
                               ↪  aadlen);

int wolfSSL_quic_aead_decrypt(uint8_t *dest, WOLFSSL_EVP_CIPHER_CTX *ctx,
                              const uint8_t *enc, size_t enclen,
                              const uint8_t *iv, const uint8_t *aad, size_t
                               ↪  aadlen);

int wolfSSL_quic_hkdf_extract(uint8_t *dest, const WOLFSSL_EVP_MD *md,
                              const uint8_t *secret, size_t secretlen,
                              const uint8_t *salt, size_t saltlen);

int wolfSSL_quic_hkdf_expand(uint8_t *dest, size_t destlen,
                             const WOLFSSL_EVP_MD *md,
                             const uint8_t *secret, size_t secretlen,
                             const uint8_t *info, size_t infolen);

int wolfSSL_quic_hkdf(uint8_t *dest, size_t destlen,
                      const WOLFSSL_EVP_MD *md,
                      const uint8_t *secret, size_t secretlen,
                      const uint8_t *salt, size_t saltlen,
                      const uint8_t *info, size_t infolen);
```

C.47 random.h

C.47.1 Functions

	Name
int	**wc_InitNetRandom**(const char * configFile, wnr_hmac_key hmac_cb, int timeout)Init global Whitewood netRandom context.
int	**wc_FreeNetRandom**(void)Free global Whitewood netRandom context.

	Name
int	**wc_InitRng**(WC_RNG * rng)Gets the seed (from OS) and key cipher for rng. rng_>drbg (deterministic random bit generator) allocated (should be deallocated with wc_FreeRng). This is a blocking operation.
int	**wc_RNG_GenerateBlock**(WC_RNG * rng, byte * output, word32 sz)Copies a sz bytes of pseudorandom data to output. Will reseed rng if needed (blocking).
int	**wc_RNG_GenerateByte**(WC_RNG * rng, byte * b)Calls wc_RNG_GenerateBlock to copy a byte of pseudorandom data to b. Will reseed rng if needed.
int	**wc_FreeRng**(WC_RNG * rng)Should be called when RNG no longer needed in order to securely free drgb. Zeros and XFREEs rng_drbg. The WC_RNG must have come from wc_InitRng*() or be zeroed.
int	**wc_RNG_HealthTest**(int reseed, const byte * seedA, word32 seedASz, const byte * seedB, word32 seedBSz, byte * output, word32 outputSz)Creates and tests functionality of drbg.
int	**wc_GenerateSeed**(OS_Seed * os, byte * output, word32 sz)Generates seed from OS entropy source. Lower-level function used internally by wc_InitRng.
WC_RNG *	**wc_rng_new**(byte * nonce, word32 nonceSz, void * heap)Allocates and initializes new WC_RNG with optional nonce.
int	**wc_rng_new_ex**(WC_RNG ** rng, byte * nonce, word32 nonceSz, void * heap, int devId)Allocates and initializes WC_RNG with extended parameters.
void	**wc_rng_free**(WC_RNG * rng)Frees WC_RNG allocated with wc_rng_new.
int	**wc_InitRng_ex**(WC_RNG * rng, void * heap, int devId)Initializes WC_RNG with extended parameters.
int	**wc_InitRngNonce**(WC_RNG * rng, const byte * nonce, word32 nonceSz)Initializes WC_RNG with nonce.
int	**wc_InitRngNonce_ex**(WC_RNG * rng, const byte * nonce, word32 nonceSz, void * heap, int devId)Initializes WC_RNG with nonce and extended parameters.
int	**wc_SetSeed_Cb**(wc_RngSeed_Cb cb)Sets callback for custom seed generation.
int	**wc_RNG_DRBG_Reseed**(WC_RNG * rng, const byte * seed, word32 seedSz)Reseeds DRBG with new entropy.

	Name
int	**wc_RNG_TestSeed**(const byte * seed, word32 seedSz)Tests seed validity for DRBG.
int	**wc_RNG_HealthTest_ex**(int reseed, const byte * nonce, word32 nonceSz, const byte * seedA, word32 seedASz, const byte * seedB, word32 seedBSz, byte * output, word32 outputSz, void * heap, int devId)RNG health test with extended parameters.
int	**wc_RNG_HealthTest_SHA512**(int reseed, const byte * seedA, word32 seedASz, const byte * seedB, word32 seedBSz, byte * output, word32 outputSz)Runs the SHA_512 Hash_DRBG Known Answer Test (KAT) per SP 800-90A. Instantiates a SHA-512 DRBG with seedA, optionally reseeds with seedB, generates output, and compares against known test vectors. Available when WOLFSSL_DRBG_SHA512 is defined.
int	**wc_RNG_HealthTest_SHA512_ex**(int reseed, const byte * nonce, word32 nonceSz, const byte * persoString, word32 persoStringSz, const byte * seedA, word32 seedASz, const byte * seedB, word32 seedBSz, const byte * additionalA, word32 additionalASz, const byte * additionalB, word32 additionalBSz, byte * output, word32 outputSz, void * heap, int devId)Extended SHA-512 Hash_DRBG health test with nonce, personalization string, and additional input support. Suitable for full ACVP / CAVP test vector validation. Available when WOLFSSL_DRBG_SHA512 is defined.
int	**wc_Sha256Drbg_Disable**(void)Disables the SHA_256 Hash_DRBG at runtime. When disabled, newly initialized WC_RNG instances will not use the SHA_256 DRBG. If the SHA_512 DRBG is enabled (WOLFSSL_DRBG_SHA512), new RNG instances will use SHA-512 instead. Requires HAVE_HASHDRBG.
int	**wc_Sha256Drbg_Enable**(void)Re_enables the SHA_256 Hash_DRBG at runtime after a prior call to wc_Sha256Drbg_Disable(). Requires HAVE_HASHDRBG.
int	**wc_Sha256Drbg_IsDisabled**(void)Returns whether the SHA-256 Hash_DRBG is currently disabled. Requires HAVE_HASHDRBG.

	Name
int	**wc_Sha512Drbg_Disable**(void)Disables the SHA-512 Hash_DRBG at runtime. When disabled, newly initialized WC_RNG instances will not use the SHA-512 DRBG. If the SHA-256 DRBG is still enabled, new RNG instances will fall back to SHA-256. Available when WOLFSSL_DRBG_SHA512 is defined. Requires HAVE_HASHDRBG.
int	**wc_Sha512Drbg_Enable**(void)Re_enables the SHA_512 Hash_DRBG at runtime after a prior call to wc_Sha512Drbg_Disable(). Available when WOLFSSL_DRBG_SHA512 is defined. Requires HAVE_HASHDRBG.
int	**wc_Sha512Drbg_IsDisabled**(void)Returns whether the SHA-512 Hash_DRBG is currently disabled. Available when WOLFSSL_DRBG_SHA512 is defined. Requires HAVE_HASHDRBG.
int	**wc_InitRng_ex2**(WC_RNG * rng, void * heap, int devId, word32 flags)Initialize a WC_RNG with instantiation_time security attributes. Identical to wc_InitRng_ex(), with a flags argument fixing attributes at birth: WC_RNG_INIT_FLAG_LOCK_REQUIRED latches the sticky lock_required policy bit, so there is no reachable state in which the instance serves without its lock policy; WC_RNG_INIT_FLAG_LOCK_INITIALLY constructs into a held lease, to be released with wc_RNG_lock_put(); WC_RNG_INIT_FLAG_USE_FULL_MUTEX layers a blocking wolfSSL_Mutex outermost around the lock latch, for user_mode sharing of one instance among threads (requires WC_RNG_HAVE_LOCK_FULL_MUTEX).
int	**wc_InitRngNonce_ex2**(WC_RNG * rng, const byte * nonce, word32 nonceSz, const byte * perso, word32 persoSz, void * heap, int devId, word32 flags)Initialize a WC_RNG with a caller_supplied nonce and instantiation_time security attributes. The nonce semantics are those of wc_InitRngNonce_ex(); the flags semantics are those of wc_InitRng_ex2().
int	**wc_RNG_GetStatus**(const WC_RNG * rng)Read_only accessor for the RNG health status. Returns the instance's enum wc_RngHealthState value (WC_DRBG_NOT_INIT, WC_DRBG_OK, WC_DRBG_FAILED, WC_DRBG_CONT_FAILED).

	Name
int	**wc_RNG_DRBG_Present**(const WC_RNG * rng)Returns 1 if rng has an instantiated DRBG, else 0. An in_service WC_RNG can lack one: instantiation bypasses the DRBG when the CPU has RDRAND (HAVE_INTEL_RDRAND). DRBG_specific services (commanded reseed, banked next seeds, RBG chains) are unavailable on such instances.
int	**wc_RNG_DRBG_GetReseedCtr**(const WC_RNG * rng, wc_drbg_reseed_ctr_t * reseedCtr)Report the DRBG's current reseed counter – the number of generate operations since the last credited (re)seed, starting at 1.
int	**wc_RNG_DRBG_ScheduleReseed**(WC_RNG * rng)Mark rng due for reseed: the next generate operation reseeds from the module's built-in or registered seed source before producing output. This can only shorten the current seed's remaining lifetime, never extend it.
int	**wc_RNG_DRBG_Reseed_Now**(WC_RNG * rng, const byte * nonce, word32 nonceSz)Immediately reseed rng from the module's built-in or registered seed source, with an optional nonce as additional input. The credited reseed resets the reseed counter.
int	**wc_RNG_DRBG_Reseed_Nonce**(WC_RNG * rng, const byte * seed, word32 seedSz, const byte * nonce, word32 nonceSz)Reseed rng's DRBG with caller-supplied seed material and an optional nonce as additional input. The material is credited as entropy: the reseed counter resets.
int	**wc_RNG_DRBG_Stir**(WC_RNG * rng, const byte * seed, word32 seedSz)Similar to wc_RNG_DRBG_Reseed(), except the caller-supplied material is mixed through the reseed derivation function without being credited as entropy: the reseed counter is not reset, so only the module's own seed source ever extends the instance's seed lifetime.
int	**wc_RNG_DRBG_Stir_Nonce**(WC_RNG * rng, const byte * seed, word32 seedSz, const byte * nonce, word32 nonceSz)The nonce_bearing form of wc_RNG_DRBG_Stir().

	Name
int	**wc_InitRngRBGC**(WC_RNG * child, WC_RNG * parent, word32 flags)Instantiate child as an SP 800_90C RBG_chain member subordinate to parent, drawing its seed material from parent's generate function in place of the module's seed source. Every other aspect of instantiation is wc_InitRng_ex2()'s. The child is tagged with stratum (parent's stratum + 1), sticky for the instance's lifetime even across subsequent source reseeds; its claimable security strength is capped by parent's, and it has no prediction resistance. The caller must hold exclusive access to parent for the duration of the call; the spawn debits parent's reseed counter by one generate.
int	**wc_InitRngNonceRBGC**(WC_RNG * child, WC_RNG * parent, const byte * nonce, word32 nonceSz, word32 flags)The nonce_bearing form of wc_InitRngRBGC(): the nonce is used as additional instantiation input, as in wc_InitRngNonce_ex2().
int	**wc_InitRngRBGC_New**(WC_RNG ** child, WC_RNG * parent, word32 flags)The allocating form of wc_InitRngRBGC(): the child is allocated from parent's heap and returned through child. Release with wc_rng_free().
int	**wc_InitRngNonceRBGC_New**(WC_RNG ** child, WC_RNG * parent, const byte * nonce, word32 nonceSz, word32 flags)The allocating, nonce_bearing form of wc_InitRngRBGC().
int	**wc_RNG_DRBG_ReseedRBGC**(WC_RNG * rng, WC_RNG * root, const byte * nonce, word32 nonceSz)Reseed rng from root's generate output – the SP 800_90C chain reseed – with an optional nonce as additional input. The reseed is credited (the reseed counter resets) and rng acquires root's stratum plus one. The caller must hold exclusive access to both instances.
int	**wc_RNG_DRBG_StirRBGC**(WC_RNG * rng, WC_RNG * root, const byte * nonce, word32 nonceSz)The uncredited form of wc_RNG_DRBG_ReseedRBGC(): material from root is mixed in without resetting rng's reseed counter.
int	**wc_RNG_DRBG_GetRBGCStratum**(const WC_RNG * rng)Report rng's RBG_chain stratum: 0 for a root (never chain_seeded), n for a member seeded from a stratum_(n_1) parent. The stratum is sticky for the instance's lifetime, even across subsequent source reseeds.

	Name
int	**wc_RNG_DRBG_GetNextSeedRBGCStratum**(const WC_RNG * rng)Report the RBG-chain stratum of rng's banked next seed – race-free via the aperture protocol – for provenance-aware consumers.
int	**wc_RNG_DRBG_NextSeedGenerate**(WC_RNG * rng, word32 n)Bank up to n more bytes of next-seed material from the module's seed source, health-testing and publishing the bank when it completes. The fill is incremental and in-boundary; a scheduling daemon may call this without owning the instance – the single-writer fill and the atomic aperture hand-off make it safe alongside a concurrent consumer.
int	**wc_RNG_DRBG_NextSeedGenerate_RBGC**(WC_RNG * rng, WC_RNG * root, word32 n)The chain_sourced form of wc_RNG_DRBG_NextSeedGenerate(): the banked material is drawn from root's generate function, and the bank is tagged with root's stratum plus one for provenance-aware consumption.
int	**wc_RNG_DRBG_NextSeedCurrent**(WC_RNG * rng, WC_ATOMIC_INT_ARG * n)Report the raw next_seed aperture value: a non_negative banked byte count (filling), WC_DRBG_NEXT_SEED_READY, or WC_DRBG_NEXT_SEED_CONSUMING. The snapshot is racy by design; use it for scheduling and diagnostics, not for hand-off decisions.
int	**wc_RNG_DRBG_NextSeedNow**(WC_RNG * rng)Claim a ready next_seed bank and perform a source_free credited reseed with it – safe in atomic context. The bank empties (use_once) and the reseed counter resets. The caller must own the instance.
int	**wc_RNG_DRBG_NextSeedNow_Nonce**(WC_RNG * rng, const byte * nonce, word32 nonceSz)The nonce_bearing form of wc_RNG_DRBG_NextSeedNow(): the nonce is mixed in as uncredited additional input alongside the banked seed.

	Name
int	**wc_RNG_DRBG_NextStirStore**(WC_RNG * rng, const byte * nonce, word32 nonceSz)Bank caller_supplied material (up to WC_RNG_NEXT_STIR_LEN bytes) in the uncredited accumulator beside the banked next seed. Writer_safe without a lease (read_copy_store); if the accumulator is already full, the material is absorbed by xor. Harvested entropy deposited here improves the instance without claiming credit.
int	**wc_RNG_DRBG_NextStirNow**(WC_RNG * rng)Stir the banked uncredited accumulator into the DRBG as an uncredited, source-free mix-in – safe in atomic context; the reseed counter is not reset. The caller must own the instance.
int	**wc_RNG_lock_get**(WC_RNG * rng, WC_RNG_lock_arg_t extra_bits)Acquire rng's exclusive-ownership lock latch, spinning on the CAS until acquired, and or the caller's extra bits into the lock word. On an instance without the lock-required policy the call is a successful no-op unless extra bits are supplied.
int	**wc_RNG_lock_get_conditional**(WC_RNG * rng, WC_RNG_lock_arg_t expected_extra_bits, WC_RNG_lock_arg_t want_extra_bits)The conditional form of wc_RNG_lock_get(): acquire only if the current extra bits equal expected_extra_bits, atomically replacing them with want_extra_bits on success. Non-blocking with respect to the condition: a mismatch fails immediately rather than spinning.
int	**wc_RNG_lock_put**(WC_RNG * rng, WC_RNG_lock_arg_t extra_bits)Release rng's lock latch, clearing the supplied extra bits atomically with the release.
int	**wc_RNG_lock_put_conditional**(WC_RNG * rng, WC_RNG_lock_arg_t expected_extra_bits, WC_RNG_lock_arg_t want_extra_bits)The conditional form of wc_RNG_lock_put(): release only if the current extra bits equal expected_extra_bits, atomically replacing them with want_extra_bits on success.
int	**wc_RNG_lock_read**(WC_RNG * rng, WC_RNG_lock_arg_t * state)Read rng's lock word: the held/required latch bits, the entropy-invalidated bit, and any caller extra bits. The snapshot is racy by design.
int	**wc_RNG_lock_set_extra**(WC_RNG * rng, WC_RNG_lock_arg_t extra_bits)Atomically set (or) the supplied caller extra bits in rng's lock word. The caller should hold the latch.

	Name
int	**wc_RNG_lock_add_extra**(WC_RNG * rng, WC_RNG_lock_arg_t extra_bits)Atomically add the supplied value to the caller extra-bits field of rng's lock word – for counters carried above WC_RNG_LOCK_EXTRA_SHIFT.
int	**wc_RNG_lock_clear_extra**(WC_RNG * rng, WC_RNG_lock_arg_t extra_bits)Atomically clear the supplied caller extra bits in rng's lock word.
int	**wc_RNG_invalidate_entropy**(WC_RNG * rng)Mark rng's seed material untrusted – for VM fork/resume and similar duplication events – opening a new invalidation epoch: banked and pooled pre_event material is purged and wiped first, then the entropy_invalidated bit is latched in the lock word. An invalidated instance refuses service (NEEDS_RECOVERY_E) until recovery_reseeded. Latch or condemn: on any error return the latch is down, and the instance is instead condemned (status DRBG_FAILED). A condemned bank instance is retired and recovered by the entropy daemon; a condemned leaf gets no daemon rescue – its owner sees RNG_FAILURE_E from subsequent operations and recovers it with wc_FreeRng() then wc_InitRng*().
int	**wc_RNG_register_free_hook**(WC_RNG * rng, wc_RNG_free_hook_cb_t free_hook, void * arg)Register a callback fired by wc_FreeRng() at teardown – for external registries (e.g. a kernel_module registry that must reach every live RNG on a VM duplication event) that need to drop their reference when the object dies.
int	**wc_RNG_Pool_Alloc**(WC_RNG * rng, word32 size)Attach a random pool to rng: a buffer of size bytes of pre_generated output, filled by wc_RNG_Pool_Collect() and drained atomic_context_safely by wc_RNG_Pool_Extract(). The pool is released with the instance.
int	**wc_RNG_Pool_Collect**(WC_RNG * rng, word32 n)Generate up to n bytes into rng's pool from rng itself. The collect/extract hand-off is arbitrated by an atomic aperture word, so a single collector is safe alongside concurrent extractors.
int	**wc_RNG_Pool_Collect2**(WC_RNG * rng_dest, WC_RNG * rng_src, word32 n)The two_instance form of wc_RNG_Pool_Collect(): fill rng_dest's pool with output drawn from rng_src – so a service instance's pool can be topped up by a daemon-owned generator.

	Name
int	**wc_RNG_Pool_Extract**(WC_RNG * rng, byte * out, word32 * n)Drain up to *n bytes from rng's pool into out – atomic-context-safe. On success* n reports the bytes actually delivered; on any error return *n is left unmodified.
int	**wc_RNG_Pool_Current**(WC_RNG * rng, word32 * n)Report the pool's current fill in bytes. The snapshot is racy by design.
int	**wc_rng_debug_stats_snap**(struct wc_rng_debug_stats_snapshot * s, const WC_RNG * rng)Snapshot the global RNG debug counters (WC_RNG_DEBUG_STATS) – seeds and reseeds by provenance, generates, pool and bank traffic – into s, for later delta accounting with wc_rng_debug_stats_sum().
int	**wc_rng_debug_stats_restore**(const struct wc_rng_debug_stats_snapshot * s, WC_RNG * rng)Restore the global RNG debug counters from a snapshot – so a test can unwind its own accounting.
int	**wc_rng_debug_stats_sum**(struct wc_rng_debug_stats_snapshot * s, const WC_RNG * rng)Accumulate the current global RNG debug counters into s – combined with a prior wc_rng_debug_stats_snap(), a delta accounting of the interval's RNG activity.

C.47.2 Functions Documentation

C.47.2.1 function wc_InitNetRandom

```
int wc_InitNetRandom(
    const char * configFile,
    wnr_hmac_key hmac_cb,
    int timeout
)
```

Init global Whitewood netRandom context.

Parameters:

- **configFile** Path to configuration file
- **hmac_cb** Optional to create HMAC callback.
- **timeout** A timeout duration.

See: wc_FreeNetRandom

Return:

- 0 Success
- BAD_FUNC_ARG Either configFile is null or timeout is negative.
- RNG_FAILURE_E There was a failure initializing the rng.

Example

```
char* config = "path/to/config/example.conf";
int time = // Some sufficient timeout value;

if (wc_InitNetRandom(config, NULL, time) != 0)
{
    // Some error occurred
}
```

C.47.2.2 function wc_FreeNetRandom

```
int wc_FreeNetRandom(
    void
)
```

Free global Whitewood netRandom context.

Parameters:

- **none** No returns.

See: wc_InitNetRandom

Return:

- 0 Success
- BAD_MUTEX_E Error locking mutex on wnr_mutex

Example

```
int ret = wc_FreeNetRandom();
if(ret != 0)
{
    // Handle the error
}
```

C.47.2.3 function wc_InitRng

```
int wc_InitRng(
    WC_RNG * rng
)
```

Gets the seed (from OS) and key cipher for rng. rng->drbg (deterministic random bit generator) allocated (should be deallocated with wc_FreeRng). This is a blocking operation.

Parameters:

- **rng** random number generator to be initialized for use with a seed and key cipher

See:

- wc_InitRngCavium
- wc_RNG_GenerateBlock
- wc_RNG_GenerateByte
- wc_FreeRng
- wc_RNG_HealthTest

Return:

- 0 on success.
- MEMORY_E XMALLOC or the fork handler registration failed
- WINCRYPT_E wc_GenerateSeed: failed to acquire context
- CRYPTGEN_E wc_GenerateSeed: failed to get random

- BAD_FUNC_ARG wc_RNG_GenerateBlock input is null or sz exceeds MAX_REQUEST_LEN
- DRBG_CONT_FIPS_E wc_RNG_GenerateBlock: Hash_gen returned DRBG_CONT_FAILURE
- ENTROPY_RT_E or ENTROPY_APT_E wc_InitRng: the SP 800-90B seed health test rejected the entropy gathered to instantiate
- RNG_FAILURE_E wc_RNG_GenerateBlock: Default error. rng's status originally not ok, or set to DRBG_FAILED
- BAD_MUTEX_E the lock that lets threads share this rng could not be created; define WC_RNG_NO_AUTO_LOCK to build without it

One WC_RNG may be shared between threads for generating and reseeding: wc_RNG_GenerateBlock(), wc_RNG_DRBG_Reseed(), wc_RNG_DRBG_Reseed_Nonce() and wc_RNG_DRBG_Reseed_Now() each hold the instance lock. So do the DRBG management calls: wc_RNG_DRBG_Stir(), wc_RNG_DRBG_Stir_Nonce() and wc_RNG_DRBG_ScheduleReseed(), and in –enable_rng_extras builds wc_RNG_DRBG_NextStirNow(), wc_RNG_DRBG_NextSeedNow() and wc_RNG_DRBG_NextSeedNow_Nonce(). The generate path calls their internal cores, which already run under the lock.

wc_RNG_DRBG_ReseedRBGC() and wc_RNG_DRBG_StirRBGC() are the exception: they draw from a second instance, so locking both would put two instance locks in play at once, and two threads reseeding each other from the other could then wait on each other for good. A caller that needs them on a shared instance stops the other threads first.

WC_RNG_NO_AUTO_LOCK (configure –disable-rng-autolock) leaves the lock out; CMSIS-RTOS v1 builds have none, its mutex pool holds ten. Every backend but a crypto callback runs with the lock held; a callback answers first, so it may fall back to the same instance. A seed callback runs with the lock held and must not use the RNG API. wc_InitRng*() and wc_FreeRng() do not lock; initialize only a new or freed WC_RNG, with no other thread using it.

POSIX lets a forked child of a threaded process only exec. Where the build has pthread_atfork(), unnamed POSIX semaphores and the dladdr() pin, fork handlers let the child keep using its WC_RNG: the parent holds every lock across fork() and the child releases them and reseeds, so each fork() waits for every live instance's generate in flight. The child should use an instance it already has: wc_InitRng() there still waits on a mutex the handlers do not cover. –disable_rng_autofork leaves them out; a user_settings build defines WC_RNG_AUTOFORK to turn them on. The child's reseed uses the configured allocator and seed source, which must therefore work after fork(). wolfCrypt_Init() or the first wc_InitRng() registers them; they can never be unregistered, so the library pins itself against dlclose(). They cover WC_RNG locks only, not clone(), vfork() or _Fork(). A fork() from inside a seed or hash callback works: that thread holds the lock, so the handler leaves it alone and the child frees it as usual. A fork() called from a signal handler can hang the caller, if the signal caught a thread part_way through taking or releasing a lock; POSIX does not allow fork() from a signal handler with handlers like these anyway. Values stay safe either way, with or without the handlers: the child reseeds on its pid change.

Builds without the handlers, macOS among them, leave a forked child only exec(): a child that calls the RNG instead blocks for good if any thread held the instance lock at fork() time. A fork() from a single threaded process is unaffected, since no lock can be held by a thread the child does not have.

Example

```
RNG  rng;
int ret;

#ifdef HAVE_CAVIUM
ret = wc_InitRngCavium(&rng, CAVIUM_DEV_ID);
if (ret != 0){
    printf("RNG Nitrox init for device: %d failed", CAVIUM_DEV_ID);
    return -1;
}
#endif
ret = wc_InitRng(&rng);
```

```
if (ret != 0){
    printf("RNG init failed");
    return -1;
}
```

C.47.2.4 function wc_RNG_GenerateBlock

```
int wc_RNG_GenerateBlock(
    WC_RNG * rng,
    byte * output,
    word32 sz
)
```

Copies a sz bytes of pseudorandom data to output. Will reseed rng if needed (blocking).

Parameters:

- **rng** random number generator initialized with wc_InitRng
- **output** buffer to which the block is copied
- **sz** size of output in bytes

See:

- wc_InitRngCavium, wc_InitRng
- wc_RNG_GenerateByte
- wc_FreeRng
- wc_RNG_HealthTest

Return:

- 0 on success
- BAD_FUNC_ARG an input is null or sz exceeds MAX_REQUEST_LEN
- DRBG_CONT_FIPS_E Hash_gen returned DRBG_CONT_FAILURE
- ENTROPY_RT_E or ENTROPY_APT_E the SP 800-90B seed health test rejected the entropy gathered for a reseed
- RNG_FAILURE_E Default error. rng's status originally not ok, or set to DRBG_FAILED
- BAD_MUTEX_E the rng's lock could not be taken

Example

```
RNG  rng;
int  sz = 32;
byte block[sz];

int ret = wc_InitRng(&rng);
if (ret != 0) {
    return -1; //init of rng failed!
}

ret = wc_RNG_GenerateBlock(&rng, block, sz);
if (ret != 0) {
    return -1; //generating block failed!
}
```

C.47.2.5 function wc_RNG_GenerateByte

```
int wc_RNG_GenerateByte(
    WC_RNG * rng,
```

```
    byte * b
)
```

Calls wc_RNG_GenerateBlock to copy a byte of pseudorandom data to b. Will reseed rng if needed.

Parameters:

- **rng** random number generator initialized with wc_InitRng
- **b** one byte buffer to which the block is copied

See:

- wc_InitRngCavium
- wc_InitRng
- wc_RNG_GenerateBlock
- wc_FreeRng
- wc_RNG_HealthTest

Return:

- 0 on success
- BAD_FUNC_ARG an input is null or sz exceeds MAX_REQUEST_LEN
- DRBG_CONT_FIPS_E Hash_gen returned DRBG_CONT_FAILURE
- RNG_FAILURE_E Default error. rng's status originally not ok, or set to DRBG_FAILED
- BAD_MUTEX_E the rng's lock could not be taken

Example

```
RNG  rng;
int  sz = 32;
byte b[1];

int ret = wc_InitRng(&rng);
if (ret != 0) {
    return -1; //init of rng failed!
}

ret = wc_RNG_GenerateByte(&rng, b);
if (ret != 0) {
    return -1; //generating block failed!
}
```

C.47.2.6 function wc_FreeRng

```
int wc_FreeRng(
    WC_RNG * rng
)
```

Should be called when RNG no longer needed in order to securely free drgb. Zeros and XFREEs rng-drbg. The WC_RNG must have come from wc_InitRng*() or be zeroed.

Parameters:

- **rng** random number generator initialized with wc_InitRng

See:

- wc_InitRngCavium
- wc_InitRng
- wc_RNG_GenerateBlock
- wc_RNG_GenerateByte,

- wc_RNG_HealthTest

Return:

- 0 on success
- BAD_FUNC_ARG rng is NULL
- RNG_FAILURE_E Failed to deallocated drbg

Example

```
RNG  rng;
int ret = wc_InitRng(&rng);
if (ret != 0) {
    return -1; //init of rng failed!
}

int ret = wc_FreeRng(&rng);
if (ret != 0) {
    return -1; //free of rng failed!
}
```

C.47.2.7 function wc_RNG_HealthTest

```
int wc_RNG_HealthTest(
    int reseed,
    const byte * seedA,
    word32 seedASz,
    const byte * seedB,
    word32 seedBSz,
    byte * output,
    word32 outputSz
)
```

Creates and tests functionality of drbg.

Parameters:

- **int** reseed: if set, will test reseed functionality
- **seedA** seed to instantiate drgb with
- **seedASz** size of seedA in bytes
- **seedB** If reseed set, drbg will be reseeded with seedB
- **seedBSz** size of seedB in bytes
- **output** initialized to random data seeded with seedB if seedrandom is set, and seedA otherwise
- **outputSz** length of output in bytes

See:

- wc_InitRngCavium
- wc_InitRng
- wc_RNG_GenerateBlock
- wc_RNG_GenerateByte
- wc_FreeRng

Return:

- 0 on success
- BAD_FUNC_ARG seedA and output must not be null. If reseed set seedB must not be null
- -1 test failed

Example

```
byte output[SHA256_DIGEST_SIZE * 4];
const byte test1EntropyB[] = ....; // test input for reseed false
const byte test1Output[] = ....;   // testvector: expected output of
                                   // reseed false
ret = wc_RNG_HealthTest(0, test1Entropy, sizeof(test1Entropy), NULL, 0,
                        output, sizeof(output));
if (ret != 0)
    return -1;//healthtest without reseed failed

if (XMEMCMP(test1Output, output, sizeof(output)) != 0)
    return -1; //compare to testvector failed: unexpected output

const byte test2EntropyB[] = ....; // test input for reseed
const byte test2Output[] = ....;   // testvector expected output of reseed
ret = wc_RNG_HealthTest(1, test2EntropyA, sizeof(test2EntropyA),
                        test2EntropyB, sizeof(test2EntropyB),
                        output, sizeof(output));

if (XMEMCMP(test2Output, output, sizeof(output)) != 0)
    return -1; //compare to testvector failed
```

C.47.2.8 function wc_GenerateSeed

```
int wc_GenerateSeed(
    OS_Seed * os,
    byte * output,
    word32 sz
)
```

Generates seed from OS entropy source. Lower-level function used internally by wc_InitRng.

Parameters:

- **os** Pointer to OS_Seed structure
- **output** Buffer to store seed
- **sz** Size of seed in bytes

See: wc_InitRng

Return:

- 0 On success
- WINCRYPT_E Failed to acquire context (Windows)
- CRYPTGEN_E Failed to generate random (Windows)
- RNG_FAILURE_E Failed to read entropy

Example

```
OS_Seed os;
byte seed[32];
int ret = wc_GenerateSeed(&os, seed, sizeof(seed));
```

C.47.2.9 function wc_rng_new

```
WC_RNG * wc_rng_new(
    byte * nonce,
    word32 nonceSz,
```

```
    void * heap
)
```

Allocates and initializes new WC_RNG with optional nonce.

Parameters:

- **nonce** Nonce buffer (can be NULL)
- **nonceSz** Nonce size
- **heap** Heap hint (can be NULL)

See: wc_rng_free

Return:

- Pointer to WC_RNG on success
- NULL on failure

Example

```
WC_RNG* rng = wc_rng_new(NULL, 0, NULL);
wc_rng_free(rng);
```

C.47.2.10 function wc_rng_new_ex

```
int wc_rng_new_ex(
    WC_RNG ** rng,
    byte * nonce,
    word32 nonceSz,
    void * heap,
    int devId
)
```

Allocates and initializes WC_RNG with extended parameters.

Parameters:

- **rng** Pointer to store WC_RNG pointer
- **nonce** Nonce buffer (can be NULL)
- **nonceSz** Nonce size
- **heap** Heap hint (can be NULL)
- **devId** Device ID (INVALID_DEVID for software)

See: wc_rng_new

Return:

- 0 On success
- BAD_FUNC_ARG If rng is NULL
- MEMORY_E Memory allocation failed
- BAD_MUTEX_E the lock that lets threads share this rng could not be created

Example

```
WC_RNG* rng;
int ret = wc_rng_new_ex(&rng, NULL, 0, NULL, INVALID_DEVID);
wc_rng_free(rng);
```

C.47.2.11 function wc_rng_free

```
void wc_rng_free(
    WC_RNG * rng
)
```

Frees WC_RNG allocated with wc_rng_new.

Parameters:

- **rng** WC_RNG to free

See: wc_rng_new

Example

```
WC_RNG* rng = wc_rng_new(NULL, 0, NULL);
wc_rng_free(rng);
```

C.47.2.12 function wc_InitRng_ex

```
int wc_InitRng_ex(
    WC_RNG * rng,
    void * heap,
    int devId
)
```

Initializes WC_RNG with extended parameters.

Parameters:

- **rng** WC_RNG to initialize
- **heap** Heap hint (can be NULL)
- **devId** Device ID (INVALID_DEVID for software)

See: wc_InitRng

Return:

- 0 On success
- BAD_FUNC_ARG If rng is NULL
- RNG_FAILURE_E Initialization failed
- BAD_MUTEX_E the lock that lets threads share this rng could not be created
- MEMORY_E the fork handlers could not be registered

Example

```
WC_RNG rng;
int ret = wc_InitRng_ex(&rng, NULL, INVALID_DEVID);
wc_FreeRng(&rng);
```

C.47.2.13 function wc_InitRngNonce

```
int wc_InitRngNonce(
    WC_RNG * rng,
    const byte * nonce,
    word32 nonceSz
)
```

Initializes WC_RNG with nonce.

Parameters:

- **rng** WC_RNG to initialize
- **nonce** Nonce buffer

- **nonceSz** Nonce size

See: wc_InitRng

Return:

- 0 On success
- BAD_FUNC_ARG If rng is NULL
- RNG_FAILURE_E Initialization failed
- BAD_MUTEX_E the lock that lets threads share this rng could not be created
- MEMORY_E the fork handlers could not be registered

Example

```
WC_RNG rng;
byte nonce[16];
int ret = wc_InitRngNonce(&rng, nonce, sizeof(nonce));
wc_FreeRng(&rng);
```

C.47.2.14 function wc_InitRngNonce_ex

```
int wc_InitRngNonce_ex(
    WC_RNG * rng,
    const byte * nonce,
    word32 nonceSz,
    void * heap,
    int devId
)
```

Initializes WC_RNG with nonce and extended parameters.

Parameters:

- **rng** WC_RNG to initialize
- **nonce** Nonce buffer
- **nonceSz** Nonce size
- **heap** Heap hint (can be NULL)
- **devId** Device ID (INVALID_DEVID for software)

See: wc_InitRngNonce

Return:

- 0 On success
- BAD_FUNC_ARG If rng is NULL
- RNG_FAILURE_E Initialization failed
- BAD_MUTEX_E the lock that lets threads share this rng could not be created
- MEMORY_E the fork handlers could not be registered

Example

```
WC_RNG rng;
byte nonce[16];
int ret = wc_InitRngNonce_ex(&rng, nonce, sizeof(nonce), NULL,
                             INVALID_DEVID);
wc_FreeRng(&rng);
```

C.47.2.15 function wc_SetSeed_Cb

```
int wc_SetSeed_Cb(
    wc_RngSeed_Cb cb
)
```

Sets callback for custom seed generation.

Parameters:

- **cb** Seed callback function

See: wc_GenerateSeed

Return:

- 0 On success
- BAD_FUNC_ARG If cb is NULL

Example

```
int my_cb(OS_Seed* os, byte* out, word32 sz) { return 0; }
wc_SetSeed_Cb(my_cb);
```

C.47.2.16 function wc_RNG_DRBG_Reseed

```
int wc_RNG_DRBG_Reseed(
    WC_RNG * rng,
    const byte * seed,
    word32 seedSz
)
```

Reseeds DRBG with new entropy.

Parameters:

- **rng** WC_RNG to reseed
- **seed** Seed buffer
- **seedSz** Seed size

See: wc_InitRng

Return:

- 0 On success
- BAD_FUNC_ARG If rng or seed is NULL
- RNG_FAILURE_E Reseed failed
- BAD_MUTEX_E the rng's lock could not be taken

Example

```
WC_RNG rng;
byte seed[32];
wc_InitRng(&rng);
int ret = wc_RNG_DRBG_Reseed(&rng, seed, sizeof(seed));
```

C.47.2.17 function wc_RNG_TestSeed

```
int wc_RNG_TestSeed(
    const byte * seed,
    word32 seedSz
)
```

Tests seed validity for DRBG.

Parameters:

- **seed** Seed to test
- **seedSz** Seed size

See: wc_InitRng

Return:

- 0 If valid
- BAD_FUNC_ARG If seed is NULL
- ENTROPY_RT_E || ENTROPY_APT_E Validation failed
- ENTROPY_APT_E The adaptive proportion test failed.

Example

```
byte seed[32];
int ret = wc_RNG_TestSeed(seed, sizeof(seed));
```

C.47.2.18 function wc_RNG_HealthTest_ex

```
int wc_RNG_HealthTest_ex(
    int reseed,
    const byte * nonce,
    word32 nonceSz,
    const byte * seedA,
    word32 seedASz,
    const byte * seedB,
    word32 seedBSz,
    byte * output,
    word32 outputSz,
    void * heap,
    int devId
)
```

RNG health test with extended parameters.

Parameters:

- **reseed** Non-zero to test reseeding
- **nonce** Nonce buffer (can be NULL)
- **nonceSz** Nonce size
- **seedA** Initial seed
- **seedASz** Initial seed size
- **seedB** Reseed buffer (required if reseed set)
- **seedBSz** Reseed size
- **output** Output buffer
- **outputSz** Output size
- **heap** Heap hint (can be NULL)
- **devId** Device ID (INVALID_DEVID for software)

See: wc_RNG_HealthTest

Return:

- 0 On success
- BAD_FUNC_ARG If required params NULL
- -1 Test failed

Example

```
byte seedA[32], seedB[32], out[64];
int ret = wc_RNG_HealthTest_ex(1, NULL, 0, seedA, 32, seedB, 32,
                               out, 64, NULL, INVALID_DEVID);
```

C.47.2.19 function wc_RNG_HealthTest_SHA512

```
int wc_RNG_HealthTest_SHA512(
    int reseed,
    const byte * seedA,
    word32 seedASz,
    const byte * seedB,
    word32 seedBSz,
    byte * output,
    word32 outputSz
)
```

Runs the SHA-512 Hash_DRBG Known Answer Test (KAT) per SP 800-90A. Instantiates a SHA-512 DRBG with seedA, optionally reseeds with seedB, generates output, and compares against known test vectors. Available when WOLFSSL_DRBG_SHA512 is defined.

Parameters:

- **reseed** Non-zero to test reseeding
- **seedA** Initial entropy seed
- **seedASz** Size of seedA in bytes
- **seedB** Reseed entropy (required if reseed is set)
- **seedBSz** Size of seedB in bytes
- **output** Buffer to receive generated output
- **outputSz** Size of output in bytes

See:

- wc_RNG_HealthTest
- wc_RNG_HealthTest_SHA512_ex

Return:

- 0 On success
- BAD_FUNC_ARG If seedA or output is NULL, or if reseed is set and seedB is NULL
- -1 Test failed

Example

```
byte output[WC_SHA512_DIGEST_SIZE * 4];
const byte seedA[] = { ... };
const byte seedB[] = { ... };

ret = wc_RNG_HealthTest_SHA512(0, seedA, sizeof(seedA), NULL, 0,
                               output, sizeof(output));
if (ret != 0)
    return -1;

ret = wc_RNG_HealthTest_SHA512(1, seedA, sizeof(seedA),
                               seedB, sizeof(seedB),
                               output, sizeof(output));
if (ret != 0)
    return -1;
```

C.47.2.20 function wc_RNG_HealthTest_SHA512_ex

```
int wc_RNG_HealthTest_SHA512_ex(
    int reseed,
    const byte * nonce,
    word32 nonceSz,
    const byte * persoString,
    word32 persoStringSz,
    const byte * seedA,
    word32 seedASz,
    const byte * seedB,
    word32 seedBSz,
    const byte * additionalA,
    word32 additionalASz,
    const byte * additionalB,
    word32 additionalBSz,
    byte * output,
    word32 outputSz,
    void * heap,
    int devId
)
```

Extended SHA-512 Hash_DRBG health test with nonce, personalization string, and additional input support. Suitable for full ACVP / CAVP test vector validation. Available when WOLFSSL_DRBG_SHA512 is defined.

Parameters:

- **reseed** Non-zero to test reseeding
- **nonce** Nonce buffer (can be NULL)
- **nonceSz** Nonce size
- **persoString** Personalization string (can be NULL)
- **persoStringSz** Personalization string size
- **seedA** Initial entropy seed
- **seedASz** Initial seed size
- **seedB** Reseed entropy (required if reseed is set)
- **seedBSz** Reseed size
- **additionalA** Additional input for first generate (can be NULL)
- **additionalASz** Additional input A size
- **additionalB** Additional input for second generate (can be NULL)
- **additionalBSz** Additional input B size
- **output** Output buffer
- **outputSz** Output size
- **heap** Heap hint (can be NULL)
- **devId** Device ID (INVALID_DEVID for software)

See:

- wc_RNG_HealthTest_SHA512
- wc_RNG_HealthTest_ex

Return:

- 0 On success
- BAD_FUNC_ARG If required params are NULL
- -1 Test failed

Example

```
byte output[WC_SHA512_DIGEST_SIZE * 4];
const byte seedA[] = { ... };
const byte nonce[] = { ... };

int ret = wc_RNG_HealthTest_SHA512_ex(0, nonce, sizeof(nonce),
                                      NULL, 0,
                                      seedA, sizeof(seedA),
                                      NULL, 0,
                                      NULL, 0, NULL, 0,
                                      output, sizeof(output),
                                      NULL, INVALID_DEVID);
```

C.47.2.21 function wc_Sha256Drbg_Disable

```
int wc_Sha256Drbg_Disable(
    void
)
```

Disables the SHA-256 Hash_DRBG at runtime. When disabled, newly initialized WC_RNG instances will not use the SHA-256 DRBG. If the SHA-512 DRBG is enabled (WOLFSSL_DRBG_SHA512), new RNG instances will use SHA-512 instead. Requires HAVE_HASHDRBG.

See:

- wc_Sha256Drbg_Enable
- wc_Sha256Drbg_IsDisabled
- wc_Sha512Drbg_Disable

Return: 0 On success

Example

```
wc_Sha256Drbg_Disable();
// New WC_RNG instances will now use SHA-512 DRBG if available
WC_RNG rng;
wc_InitRng(&rng);
```

C.47.2.22 function wc_Sha256Drbg_Enable

```
int wc_Sha256Drbg_Enable(
    void
)
```

Re-enables the SHA-256 Hash_DRBG at runtime after a prior call to wc_Sha256Drbg_Disable(). Requires HAVE_HASHDRBG.

See:

- wc_Sha256Drbg_Disable
- wc_Sha256Drbg_IsDisabled

Return: 0 On success

Example

```
wc_Sha256Drbg_Disable();
// ... use SHA-512 DRBG only ...
wc_Sha256Drbg_Enable();
// New WC_RNG instances can use SHA-256 DRBG again
```

C.47.2.23 function wc_Sha256Drbg_IsDisabled

```
int wc_Sha256Drbg_IsDisabled(
    void
)
```

Returns whether the SHA-256 Hash_DRBG is currently disabled. Requires HAVE_HASHDRBG.

See:

- wc_Sha256Drbg_Disable
- wc_Sha256Drbg_Enable

Return:

- 1 SHA-256 DRBG is disabled
- 0 SHA-256 DRBG is enabled (not disabled)

Example

```
if (wc_Sha256Drbg_IsDisabled()) {
    printf("SHA-256 DRBG is off\n");
}
```

C.47.2.24 function wc_Sha512Drbg_Disable

```
int wc_Sha512Drbg_Disable(
    void
)
```

Disables the SHA-512 Hash_DRBG at runtime. When disabled, newly initialized WC_RNG instances will not use the SHA-512 DRBG. If the SHA-256 DRBG is still enabled, new RNG instances will fall back to SHA-256. Available when WOLFSSL_DRBG_SHA512 is defined. Requires HAVE_HASHDRBG.

See:

- wc_Sha512Drbg_Enable
- wc_Sha512Drbg_IsDisabled
- wc_Sha256Drbg_Disable

Return: 0 On success

Example

```
wc_Sha512Drbg_Disable();
// New WC_RNG instances will now use SHA-256 DRBG
WC_RNG rng;
wc_InitRng(&rng);
```

C.47.2.25 function wc_Sha512Drbg_Enable

```
int wc_Sha512Drbg_Enable(
    void
)
```

Re-enables the SHA-512 Hash_DRBG at runtime after a prior call to wc_Sha512Drbg_Disable(). Available when WOLFSSL_DRBG_SHA512 is defined. Requires HAVE_HASHDRBG.

See:

- wc_Sha512Drbg_Disable
- wc_Sha512Drbg_IsDisabled

Return: 0 On success

Example

```
wc_Sha512Drbg_Disable();
// ... use SHA-256 DRBG only ...
wc_Sha512Drbg_Enable();
// New WC_RNG instances can use SHA-512 DRBG again
```

C.47.2.26 function wc_Sha512Drbg_IsDisabled

```
int wc_Sha512Drbg_IsDisabled(
    void
)
```

Returns whether the SHA-512 Hash_DRBG is currently disabled. Available when WOLFSSL_DRBG_SHA512 is defined. Requires HAVE_HASHDRBG.

See:

- wc_Sha512Drbg_Disable
- wc_Sha512Drbg_Enable

Return:

- 1 SHA-512 DRBG is disabled
- 0 SHA-512 DRBG is enabled (not disabled)

Example

```
if (wc_Sha512Drbg_IsDisabled()) {
    printf("SHA-512 DRBG is off\n");
}
```

C.47.2.27 function wc_InitRng_ex2

```
int wc_InitRng_ex2(
    WC_RNG * rng,
    void * heap,
    int devId,
    word32 flags
)
```

Initialize a WC_RNG with instantiation-time security attributes. Identical to wc_InitRng_ex(), with a flags argument fixing attributes at birth: WC_RNG_INIT_FLAG_LOCK_REQUIRED latches the sticky lock_required policy bit, so there is no reachable state in which the instance serves without its lock policy; WC_RNG_INIT_FLAG_LOCK_INITIALLY constructs into a held lease, to be released with wc_RNG_lock_put(); WC_RNG_INIT_FLAG_USE_FULL_MUTEX layers a blocking wolfSSL_Mutex outermost around the lock latch, for user_mode sharing of one instance among threads (requires WC_RNG_HAVE_LOCK_FULL_MUTEX).

Parameters:

- **rng** The RNG object to initialize.
- **heap** Heap hint for dynamic allocation.
- **devId** Device id, or INVALID_DEVID.
- **flags** Bitwise-or of WC_RNG_INIT_FLAG_* attributes.

See:

- wc_InitRng_ex

- wc_InitRngNonce_ex2
- wc_RNG_lock_get
- wc_RNG_lock_put

Return:

- 0 Success
- BAD_FUNC_ARG rng is null, or the flags contradict each other.
- NOT_COMPILED_IN A requested flag is not compiled in.

WC_RNG_INIT_FLAG_USE_AUTO_LOCK and WC_RNG_INIT_FLAG_NO_AUTO_LOCK decide, for this instance alone, whether the automatic per-call lock is created, whatever –enable-rng-autolock settled for the build. Use the first when the instance will be shared between threads: a build without that lock refuses with NOT_COMPILED_IN rather than returning an instance the caller would wrongly believe is serialized. Use the second to drop the per-call cost on an instance one thread owns; such an instance behaves as it does with the lock left out of the build, so sharing it between threads is a data race and, where the fork handlers exist, it is not fork covered. Setting both, or the first alongside WC_RNG_INIT_FLAG_USE_FULL_MUTEX, is BAD_FUNC_ARG in every build.

Naming neither leaves it to the build: on by default, or off where –disable-rng-autolock-default (WC_RNG_AUTO_LOCK_DEFAULT_OFF) built the lock in without switching it on. In such a build only the two flag forms, wc_InitRng_ex2() and wc_InitRngNonce_ex2(), can ask for the lock; wc_InitRng(), wc_InitRng_ex(), wc_rng_new() and wc_rng_new_ex() take no flags, so their instances go unlocked and cannot opt in.

Example

```
WC_RNG rng;
if (wc_InitRng_ex2(&rng, NULL, INVALID_DEVID,
                   WC_RNG_INIT_FLAG_LOCK_REQUIRED |
                   WC_RNG_INIT_FLAG_LOCK_INITIALLY) != 0) {
    // error handling
}
// caller holds the lease from birth
```

C.47.2.28 function wc_InitRngNonce_ex2

```
int wc_InitRngNonce_ex2(
    WC_RNG * rng,
    const byte * nonce,
    word32 nonceSz,
    const byte * perso,
    word32 persoSz,
    void * heap,
    int devId,
    word32 flags
)
```

Initialize a WC_RNG with a caller-supplied nonce and instantiation-time security attributes. The nonce semantics are those of wc_InitRngNonce_ex(); the flags semantics are those of wc_InitRng_ex2().

Parameters:

- **rng** The RNG object to initialize.
- **nonce** Optional nonce used as additional instantiation input.
- **nonceSz** Length of nonce in bytes.
- **heap** Heap hint for dynamic allocation.
- **devId** Device id, or INVALID_DEVID.
- **perso** Optional personalization string (may be null).

- **persoSz** Length of perso in bytes.
- **flags** Bitwise-or of WC_RNG_INIT_FLAG_* attributes.

See:

- wc_InitRng_ex2
- wc_InitRngNonce_ex

Return:

- 0 Success
- BAD_FUNC_ARG rng is null.
- NOT_COMPILED_IN A requested flag is not compiled in.

C.47.2.29 function wc_RNG_GetStatus

```
int wc_RNG_GetStatus(
    const WC_RNG * rng
)
```

Read-only accessor for the RNG health status. Returns the instance's enum wc_RngHealthState value (WC_DRBG_NOT_INIT, WC_DRBG_OK, WC_DRBG_FAILED, WC_DRBG_CONT_FAILED).

Parameters:

- **rng** The RNG object to interrogate.

See:

- wc_RNG_DRBG_Present
- wc_RNG_DRBG_GetReseedCtr

Return:

- WC_DRBG_OK The instance is in service.
- BAD_FUNC_ARG rng is null.

Example

```
if (wc_RNG_GetStatus(&rng) != WC_DRBG_OK) {
    // instance is not serviceable
}
```

C.47.2.30 function wc_RNG_DRBG_Present

```
int wc_RNG_DRBG_Present(
    const WC_RNG * rng
)
```

Returns 1 if rng has an instantiated DRBG, else 0. An in-service WC_RNG can lack one: instantiation bypasses the DRBG when the CPU has RDRAND (HAVE_INTEL_RDRAND). DRBG-specific services (commanded reseed, banked next seeds, RBG chains) are unavailable on such instances.

Parameters:

- **rng** The RNG object to interrogate.

See:

- wc_RNG_GetStatus
- wc_RNG_DRBG_ScheduleReseed

Return:

- 1 rng has a live DRBG.
- 0 rng is null or has no DRBG.

C.47.2.31 function wc_RNG_DRBG_GetReseedCtr

```
int wc_RNG_DRBG_GetReseedCtr(
    const WC_RNG * rng,
    wc_drbg_reseed_ctr_t * reseedCtr
)
```

Report the DRBG's current reseed counter – the number of generate operations since the last credited (re)seed, starting at 1.

Parameters:

- **rng** The RNG object to interrogate.
- **reseedCtr** Receives the counter.

See:

- wc_RNG_DRBG_Present
- wc_RNG_DRBG_ScheduleReseed

Return:

- 0 Success
- BAD_FUNC_ARG rng or reseedCtr is null.
- WRONG_TYPE_OBJECT_E rng has no DRBG (RDRAND et al.).

C.47.2.32 function wc_RNG_DRBG_ScheduleReseed

```
int wc_RNG_DRBG_ScheduleReseed(
    WC_RNG * rng
)
```

Mark rng due for reseed: the next generate operation reseeds from the module's built-in or registered seed source before producing output. This can only shorten the current seed's remaining lifetime, never extend it.

Parameters:

- **rng** The RNG object to schedule.

See:

- wc_RNG_DRBG_Reseed_Now
- wc_RNG_DRBG_GetReseedCtr

Return:

- 0 Success
- BAD_FUNC_ARG rng is null.
- WRONG_TYPE_OBJECT_E rng has no DRBG (RDRAND et al.) – a commanded reseed that cannot happen is not a success.

C.47.2.33 function wc_RNG_DRBG_Reseed_Now

```
int wc_RNG_DRBG_Reseed_Now(
    WC_RNG * rng,
    const byte * nonce,
```

```
    word32 nonceSz
)
```

Immediately reseed rng from the module's built-in or registered seed source, with an optional nonce as additional input. The credited reseed resets the reseed counter.

Parameters:

- **rng** The RNG object to reseed.
- **nonce** Optional additional input.
- **nonceSz** Length of nonce in bytes.

See:

- wc_RNG_DRBG_ScheduleReseed
- wc_RNG_DRBG_Reseed_Nonce

Return:

- 0 Success
- BAD_FUNC_ARG rng is null, or nonce is null with nonceSz nonzero.
- WRONG_TYPE_OBJECT_E rng has no DRBG (RDRAND et al.).
- DRBG_CONT_FIPS_E The continuous test failed; the DRBG is out of service.
- RNG_FAILURE_E The DRBG is out of service or reseeding failed.
- ENTROPY_RT_E or ENTROPY_APT_E The SP 800-90B seed health test rejected the gathered seed; the reseed did not happen and the instance remains usable until its reseed interval is reached.

C.47.2.34 function wc_RNG_DRBG_Reseed_Nonce

```
int wc_RNG_DRBG_Reseed_Nonce(
    WC_RNG * rng,
    const byte * seed,
    word32 seedSz,
    const byte * nonce,
    word32 nonceSz
)
```

Reseed rng's DRBG with caller-supplied seed material and an optional nonce as additional input. The material is credited as entropy: the reseed counter resets.

Parameters:

- **rng** The RNG object to reseed.
- **seed** Seed material.
- **seedSz** Length of seed in bytes.
- **nonce** Optional additional input.
- **nonceSz** Length of nonce in bytes.

See:

- wc_RNG_DRBG_Reseed
- wc_RNG_DRBG_Stir_Nonce

Return:

- 0 Success
- RNG_FAILURE_E rng is condemned (status DRBG_FAILED): a condemned instance does not accept a credited reseed; recover with wc_FreeRng() then wc_InitRng*().
- BAD_FUNC_ARG rng or seed is null.
- WRONG_TYPE_OBJECT_E rng has no DRBG (RDRAND et al.).

C.47.2.35 function wc_RNG_DRBG_Stir

```
int wc_RNG_DRBG_Stir(
    WC_RNG * rng,
    const byte * seed,
    word32 seedSz
)
```

Similar to wc_RNG_DRBG_Reseed(), except the caller-supplied material is mixed through the reseed derivation function without being credited as entropy: the reseed counter is not reset, so only the module's own seed source ever extends the instance's seed lifetime.

Parameters:

- **rng** The RNG object to stir.
- **seed** Material to mix in.
- **seedSz** Length of seed in bytes.

See:

- wc_RNG_DRBG_Reseed
- wc_RNG_DRBG_Stir_Nonce

Return:

- 0 Success
- BAD_FUNC_ARG rng or seed is null.
- WRONG_TYPE_OBJECT_E rng has no DRBG (RDRAND et al.).

C.47.2.36 function wc_RNG_DRBG_Stir_Nonce

```
int wc_RNG_DRBG_Stir_Nonce(
    WC_RNG * rng,
    const byte * seed,
    word32 seedSz,
    const byte * nonce,
    word32 nonceSz
)
```

The nonce-bearing form of wc_RNG_DRBG_Stir().

Parameters:

- **rng** The RNG object to stir.
- **seed** Material to mix in.
- **seedSz** Length of seed in bytes.
- **nonce** Optional additional input.
- **nonceSz** Length of nonce in bytes.

See:

- wc_RNG_DRBG_Stir
- wc_RNG_DRBG_Reseed_Nonce

Return:

- 0 Success
- BAD_FUNC_ARG rng or seed is null.
- WRONG_TYPE_OBJECT_E rng has no DRBG (RDRAND et al.).

C.47.2.37 function wc_InitRngRBGC

```
int wc_InitRngRBGC(
    WC_RNG * child,
    WC_RNG * parent,
    word32 flags
)
```

Instantiate child as an SP 800-90C RBG-chain member subordinate to parent, drawing its seed material from parent's generate function in place of the module's seed source. Every other aspect of instantiation is wc_InitRng_ex2()'s. The child is tagged with stratum (parent's stratum + 1), sticky for the instance's lifetime even across subsequent source reseeds; its claimable security strength is capped by parent's, and it has no prediction resistance. The caller must hold exclusive access to parent for the duration of the call; the spawn debits parent's reseed counter by one generate.

Parameters:

- **child** The caller-provided WC_RNG to instantiate (uninitialized).
- **parent** The chain parent to draw seed material from.
- **flags** Bitwise-or of WC_RNG_INIT_FLAG_* attributes for the child.

See:

- wc_InitRngNonceRBGC
- wc_InitRngRBGC_New
- wc_RNG_DRBG_ReseedRBGC
- wc_RNG_DRBG_GetRBGCStratum

Return:

- 0 Success
- BAD_FUNC_ARG child or parent is null, or child equals parent.
- SEQ_OVERFLOW_E parent's stratum is at the representable maximum.

Example

```
WC_RNG root, child;
wc_InitRng(&root);
if (wc_InitRngRBGC(&child, &root, WC_RNG_INIT_FLAG_NONE) == 0) {
    // child serves independently; release with wc_FreeRng(&child)
}
```

C.47.2.38 function wc_InitRngNonceRBGC

```
int wc_InitRngNonceRBGC(
    WC_RNG * child,
    WC_RNG * parent,
    const byte * nonce,
    word32 nonceSz,
    word32 flags
)
```

The nonce-bearing form of wc_InitRngRBGC(): the nonce is used as additional instantiation input, as in wc_InitRngNonce_ex2().

Parameters:

- **child** The caller-provided WC_RNG to instantiate (uninitialized).
- **parent** The chain parent to draw seed material from.
- **nonce** Additional instantiation input.
- **nonceSz** Length of nonce in bytes.

- **flags** Bitwise-or of WC_RNG_INIT_FLAG_* attributes for the child.

See:

- wc_InitRngRBGC
- wc_InitRngNonceRBGC_New

Return:

- 0 Success
- BAD_FUNC_ARG child or parent is null, child equals parent, or nonce is null with nonceSz nonzero.
- SEQ_OVERFLOW_E parent's stratum is at the representable maximum.

C.47.2.39 function wc_InitRngRBGC_New

```
int wc_InitRngRBGC_New(
    WC_RNG ** child,
    WC_RNG * parent,
    word32 flags
)
```

The allocating form of wc_InitRngRBGC(): the child is allocated from parent's heap and returned through child. Release with wc_rng_free().

Parameters:

- **child** Receives the allocated, instantiated WC_RNG.
- **parent** The chain parent to draw seed material from.
- **flags** Bitwise-or of WC_RNG_INIT_FLAG_* attributes for the child.

See:

- wc_InitRngRBGC
- wc_InitRngNonceRBGC_New

Return:

- 0 Success
- BAD_FUNC_ARG child or parent is null.
- MEMORY_E Allocation failed.
- SEQ_OVERFLOW_E parent's stratum is at the representable maximum.

C.47.2.40 function wc_InitRngNonceRBGC_New

```
int wc_InitRngNonceRBGC_New(
    WC_RNG ** child,
    WC_RNG * parent,
    const byte * nonce,
    word32 nonceSz,
    word32 flags
)
```

The allocating, nonce-bearing form of wc_InitRngRBGC().

Parameters:

- **child** Receives the allocated, instantiated WC_RNG.
- **parent** The chain parent to draw seed material from.
- **nonce** Additional instantiation input.
- **nonceSz** Length of nonce in bytes.
- **flags** Bitwise-or of WC_RNG_INIT_FLAG_* attributes for the child.

See:

- wc_InitRngRBGC_New
- wc_InitRngNonceRBGC

Return:

- 0 Success
- BAD_FUNC_ARG child or parent is null, or nonce is null with nonceSz nonzero.
- MEMORY_E Allocation failed.
- SEQ_OVERFLOW_E parent's stratum is at the representable maximum.

C.47.2.41 function wc_RNG_DRBG_ReseedRBGC

```
int wc_RNG_DRBG_ReseedRBGC(
    WC_RNG * rng,
    WC_RNG * root,
    const byte * nonce,
    word32 nonceSz
)
```

Reseed rng from root's generate output – the SP 800-90C chain reseed – with an optional nonce as additional input. The reseed is credited (the reseed counter resets) and rng acquires root's stratum plus one. The caller must hold exclusive access to both instances.

Parameters:

- **rng** The chain member to reseed.
- **root** The chain parent to draw seed material from.
- **nonce** Optional additional input.
- **nonceSz** Length of nonce in bytes.

See:

- wc_InitRngRBGC
- wc_RNG_DRBG_StirRBGC
- wc_RNG_DRBG_Reseed_Now

Return:

- 0 Success
- BAD_FUNC_ARG rng or root is null, rng equals root, or the no-downgrade rule refuses root as a chain parent (see
- WRONG_TYPE_OBJECT_E rng has no DRBG (RDRAND et al.).
- SEQ_OVERFLOW_E root's stratum is at the representable maximum.

Credited chain reseeds obey a no-downgrade rule: a primary-seeded (stratum-0) root is always accepted, and a chained (stratum > 0) root is accepted only when its stratum is strictly less than rng's – the acquired stratum never increases, so reseed cycles are impossible by construction, consistent with SP 800-90C 7.1.2.2. Lateral (equal-stratum) and downgrading reseeds are refused with BAD_FUNC_ARG. Building WC_RNG_NO_RBGC_RESEED restricts credited chain reseeds to primary-seeded roots. Uncredited chain stirs (wc_RNG_DRBG_StirRBGC()) are exempt from all of this: they are stirs, claim nothing, and leave rng's stratum untouched.

).

C.47.2.42 function wc_RNG_DRBG_StirRBGC

```
int wc_RNG_DRBG_StirRBGC(
    WC_RNG * rng,
```

```
    WC_RNG * root,
    const byte * nonce,
    word32 nonceSz
)
```

The uncredited form of wc_RNG_DRBG_ReseedRBGC(): material from root is mixed in without resetting rng's reseed counter.

Parameters:

- **rng** The chain member to stir.
- **root** The chain parent to draw material from.
- **nonce** Optional additional input.
- **nonceSz** Length of nonce in bytes.

See:

- wc_RNG_DRBG_ReseedRBGC
- wc_RNG_DRBG_Stir

Return:

- 0 Success
- BAD_FUNC_ARG rng or root is null, or rng equals root.
- WRONG_TYPE_OBJECT_E rng has no DRBG (RDRAND et al.).

Unrestricted by the credited no-downgrade rule: any source stratum is accepted, and rng's reseed counter, stratum, and entropy-invalidated state are all left untouched – an uncredited chain reseed is a stir, and a stir must never masquerade as recovery or promotion.

C.47.2.43 function wc_RNG_DRBG_GetRBGCStratum

```
int wc_RNG_DRBG_GetRBGCStratum(
    const WC_RNG * rng
)
```

Report rng's RBG-chain stratum: 0 for a root (never chain-seeded), n for a member seeded from a stratum-(n-1) parent. The stratum is sticky for the instance's lifetime, even across subsequent source reseeds.

Parameters:

- **rng** The RNG object to interrogate.

See:

- wc_InitRngRBGC
- wc_RNG_DRBG_GetNextSeedRBGCStratum

Return:

- 0 rng is a chain root.
- n The stratum, positive for a chain member.
- BAD_FUNC_ARG rng is null.

C.47.2.44 function wc_RNG_DRBG_GetNextSeedRBGCStratum

```
int wc_RNG_DRBG_GetNextSeedRBGCStratum(
    const WC_RNG * rng
)
```

Report the RBG-chain stratum of rng's banked next seed – race-free via the aperture protocol – for provenance-aware consumers.

Parameters:

- **rng** The RNG object to interrogate.

See:

- wc_RNG_DRBG_GetRBGCStratum
- wc_RNG_DRBG_NextSeedGenerate_RBGC

Return:

- 0 The banked seed has root (source) provenance.
- n The banked seed's stratum, positive for chain provenance.
- BAD_FUNC_ARG rng is null or has no DRBG.
- NOT_READY_E No banked seed is ready.

C.47.2.45 function wc_RNG_DRBG_NextSeedGenerate

```
int wc_RNG_DRBG_NextSeedGenerate(
    WC_RNG * rng,
    word32 n
)
```

Bank up to n more bytes of next-seed material from the module's seed source, health-testing and publishing the bank when it completes. The fill is incremental and in-boundary; a scheduling daemon may call this without owning the instance – the single-writer fill and the atomic aperture hand-off make it safe alongside a concurrent consumer.

Parameters:

- **rng** The RNG object whose bank to fill.
- **n** Maximum bytes to bank this call (clamped to space remaining).

See:

- wc_RNG_DRBG_NextSeedNow
- wc_RNG_DRBG_NextSeedCurrent
- wc_RNG_DRBG_NextSeedGenerate_RBGC

Return:

- 0 Bytes were banked (bank may or may not yet be complete).
- ALREADY_E The bank is ready or being consumed.
- ENTROPY_RT_E or ENTROPY_APT_E The SP 800-90B seed health test rejected the banked material, which is burned.
- BAD_FUNC_ARG rng is null or n is 0.
- MISSING_RNG_E rng has no DRBG (RDRAND et al.).

Example

```
// scheduling daemon: fill incrementally until published
int ret = wc_RNG_DRBG_NextSeedGenerate(rng, 16);
if (ret == ALREADY_E) {
    // bank is ready; nothing to do until a consumer claims it
}
```

C.47.2.46 function wc_RNG_DRBG_NextSeedGenerate_RBGC

```
int wc_RNG_DRBG_NextSeedGenerate_RBGC(
    WC_RNG * rng,
    WC_RNG * root,
    word32 n
)
```

The chain-sourced form of wc_RNG_DRBG_NextSeedGenerate(): the banked material is drawn from root's generate function, and the bank is tagged with root's stratum plus one for provenance-aware consumption.

Parameters:

- **rng** The RNG object whose bank to fill.
- **root** The chain parent to draw material from.
- **n** Maximum bytes to bank this call.

See:

- wc_RNG_DRBG_NextSeedGenerate
- wc_RNG_DRBG_GetNextSeedRBGCStratum

Return:

- 0 Bytes were banked.
- ALREADY_E The bank is ready or being consumed.
- ENTROPY_RT_E or ENTROPY_APT_E The SP 800-90B seed health test rejected the banked material, which is burned.
- BAD_FUNC_ARG rng or root is null, or n is 0.
- MISSING_RNG_E rng has no DRBG (RDRAND et al.).
- SEQ_OVERFLOW_E root's stratum is at the representable maximum.

Banking is bound for credited redemption, so the credited no-downgrade rule applies at bank time: a primary-seeded (stratum-0) root is always accepted, and a chained root only when its stratum is strictly less than rng's – banking whose redemption would raise rng's stratum is refused with BAD_FUNC_ARG. The banked material records root's stratum plus one, observable via wc_RNG_DRBG_GetNextSeedRBGCStratum(), and redemption (wc_RNG_DRBG_NextSeedNow()) carries it onto rng.

C.47.2.47 function wc_RNG_DRBG_NextSeedCurrent

```
int wc_RNG_DRBG_NextSeedCurrent(
    WC_RNG * rng,
    WC_ATOMIC_INT_ARG * n
)
```

Report the raw next-seed aperture value: a non-negative banked byte count (filling), WC_DRBG_NEXT_SEED_READY, or WC_DRBG_NEXT_SEED_CONSUMING. The snapshot is racy by design; use it for scheduling and diagnostics, not for hand-off decisions.

Parameters:

- **rng** The RNG object to interrogate.
- **n** Receives the aperture value.

See:

- wc_RNG_DRBG_NextSeedGenerate
- wc_RNG_DRBG_NextSeedNow

Return:

- 0 Success
- BAD_FUNC_ARG rng or n is null.
- MISSING_RNG_E rng has no DRBG (RDRAND et al.).

C.47.2.48 function wc_RNG_DRBG_NextSeedNow

```
int wc_RNG_DRBG_NextSeedNow(
    WC_RNG * rng
)
```

Claim a ready next-seed bank and perform a source-free credited reseed with it – safe in atomic context. The bank empties (use-once) and the reseed counter resets. The caller must own the instance.

Parameters:

- **rng** The RNG object to reseed.

See:

- wc_RNG_DRBG_NextSeedGenerate
- wc_RNG_DRBG_NextSeedNow_Nonce

Return:

- 0 Success
- NOT_READY_E No bank is ready.
- BAD_FUNC_ARG rng is null.
- MISSING_RNG_E rng has no DRBG (RDRAND et al.).
- ENTROPY_RT_E or ENTROPY_APT_E A seed health test failed on the reseed this performs; the instance is condemned.

Example

```
// atomic-context consumer
if (wc_RNG_DRBG_NextSeedNow(rng) == 0) {
    // freshly reseeded without touching the seed source
}
```

C.47.2.49 function wc_RNG_DRBG_NextSeedNow_Nonce

```
int wc_RNG_DRBG_NextSeedNow_Nonce(
    WC_RNG * rng,
    const byte * nonce,
    word32 nonceSz
)
```

The nonce-bearing form of wc_RNG_DRBG_NextSeedNow(): the nonce is mixed in as uncredited additional input alongside the banked seed.

Parameters:

- **rng** The RNG object to reseed.
- **nonce** Additional input.
- **nonceSz** Length of nonce in bytes.

See: wc_RNG_DRBG_NextSeedNow

Return:

- 0 Success
- NEEDS_RECOVERY_E A purge crossed the consume (an invalidation epoch boundary): no material is adopted, the entropy-invalidated latch is re-asserted, and a recovery reseed is scheduled.

- NOT_READY_E No bank is ready.
- BAD_FUNC_ARG rng is null, or nonce is null with nonceSz nonzero.
- MISSING_RNG_E rng has no DRBG (RDRAND et al.).
- ENTROPY_RT_E or ENTROPY_APT_E A seed health test failed on the reseed this performs; the instance is condemned.
- DRBG_CONT_FIPS_E The continuous test failed; the DRBG is out of service.
- RNG_FAILURE_E The DRBG is out of service.

C.47.2.50 function wc_RNG_DRBG_NextStirStore

```
int wc_RNG_DRBG_NextStirStore(
    WC_RNG * rng,
    const byte * nonce,
    word32 nonceSz
)
```

Bank caller-supplied material (up to WC_RNG_NEXT_STIR_LEN bytes) in the uncredited accumulator beside the banked next seed. Writer-safe without a lease (read-copy-store); if the accumulator is already full, the material is absorbed by xor. Harvested entropy deposited here improves the instance without claiming credit.

Parameters:

- **rng** The RNG object whose accumulator to feed.
- **nonce** Material to bank.
- **nonceSz** Length of nonce in bytes.

See:

- wc_RNG_DRBG_NextStirNow
- wc_RNG_DRBG_Stir

Return:

- 0 Success
- BAD_FUNC_ARG rng or nonce is null, or nonceSz is 0.
- MISSING_RNG_E rng has no DRBG (RDRAND et al.).

C.47.2.51 function wc_RNG_DRBG_NextStirNow

```
int wc_RNG_DRBG_NextStirNow(
    WC_RNG * rng
)
```

Stir the banked uncredited accumulator into the DRBG as an uncredited, source-free mix-in – safe in atomic context; the reseed counter is not reset. The caller must own the instance.

Parameters:

- **rng** The RNG object to stir.

See: wc_RNG_DRBG_NextStirStore

Return:

- 0 Success
- NOT_READY_E The accumulator is empty or still accumulating, or the stir is refused (reseed interval, entropy-invalidated quarantine).
- BUSY_E The accumulator was claimed by a racing consumer – the stir is happening by another hand.
- BAD_FUNC_ARG rng is null.

- MISSING_RNG_E rng has no DRBG (RDRAND et al.).
- RNG_FAILURE_E The DRBG is out of service, or its hash failed mid-stir leaving a half-applied update – the instance is then condemned (status DRBG_FAILED).

C.47.2.52 function wc_RNG_lock_get

```
int wc_RNG_lock_get(
    WC_RNG * rng,
    WC_RNG_lock_arg_t extra_bits
)
```

Acquire rng's exclusive-ownership lock latch, spinning on the CAS until acquired, and or the caller's extra bits into the lock word. On an instance without the lock-required policy the call is a successful no-op unless extra bits are supplied.

Parameters:

- **rng** The RNG object to lock.
- **extra_bits** Caller-defined bits (above WC_RNG_LOCK_EXTRA_SHIFT) to set atomically with the acquisition, or 0.

See:

- wc_RNG_lock_put
- wc_RNG_lock_get_conditional
- wc_RNG_lock_read

Return:

- 0 Success
- BAD_FUNC_ARG rng is null.
- BUSY_E The lock is held.
- NEEDS_RECOVERY_E The instance's entropy is invalidated (see wc_RNG_invalidate_entropy()); recover with a credited reseed before use.
- BAD_MUTEX_E (WC_RNG_HAVE_LOCK_FULL_MUTEX) The outer mutex failed.
- UNEXPECTED_STATE_E Spurious acquisition failure; retry.

Example

```
if (wc_RNG_lock_get(rng, 0) == 0) {
    ret = wc_RNG_GenerateBlock(rng, out, sizeof(out));
    wc_RNG_lock_put(rng, 0);
}
```

C.47.2.53 function wc_RNG_lock_get_conditional

```
int wc_RNG_lock_get_conditional(
    WC_RNG * rng,
    WC_RNG_lock_arg_t expected_extra_bits,
    WC_RNG_lock_arg_t want_extra_bits
)
```

The conditional form of wc_RNG_lock_get(): acquire only if the current extra bits equal expected_extra_bits, atomically replacing them with want_extra_bits on success. Non-blocking with respect to the condition: a mismatch fails immediately rather than spinning.

Parameters:

- **rng** The RNG object to lock.
- **expected_extra_bits** The extra bits required for acquisition.

- **want_extra_bits** The extra bits to install on acquisition.

See:

- wc_RNG_lock_get
- wc_RNG_lock_put_conditional

Return:

- 0 Success
- BAD_FUNC_ARG rng is null.
- BUSY_E The lock is held, or the extra bits do not match expected_extra_bits.
- NEEDS_RECOVERY_E Entropy-invalidated and WC_RNG_LOCK_ENTROPY_INVALIDATED is not in expected_extra_bits.
- BAD_MUTEX_E (WC_RNG_HAVE_LOCK_FULL_MUTEX) The outer mutex failed.
- UNEXPECTED_STATE_E Spurious acquisition failure; retry.

C.47.2.54 function wc_RNG_lock_put

```
int wc_RNG_lock_put(
    WC_RNG * rng,
    WC_RNG_lock_arg_t extra_bits
)
```

Release rng's lock latch, clearing the supplied extra bits atomically with the release.

Parameters:

- **rng** The RNG object to unlock.
- **extra_bits** Caller-defined bits to clear with the release, or 0.

See:

- wc_RNG_lock_get
- wc_RNG_lock_put_conditional

Return:

- 0 Success
- BAD_FUNC_ARG rng is null.
- OBJECT_NOT_LOCKED_E The latch is not held.
- NEEDS_RECOVERY_E Released successfully; informational notice that the instance is entropy-invalidated.

C.47.2.55 function wc_RNG_lock_put_conditional

```
int wc_RNG_lock_put_conditional(
    WC_RNG * rng,
    WC_RNG_lock_arg_t expected_extra_bits,
    WC_RNG_lock_arg_t want_extra_bits
)
```

The conditional form of wc_RNG_lock_put(): release only if the current extra bits equal expected_extra_bits, atomically replacing them with want_extra_bits on success.

Parameters:

- **rng** The RNG object to unlock.
- **expected_extra_bits** The extra bits required for release.
- **want_extra_bits** The extra bits to install on release.

See:

- wc_RNG_lock_put
- wc_RNG_lock_get_conditional

Return:

- 0 Success
- BAD_FUNC_ARG rng is null.
- BUSY_E The extra bits did not match expected_extra_bits.
- OBJECT_NOT_LOCKED_E The latch is not held.
- NEEDS_RECOVERY_E Released successfully; informational notice that the instance is entropy-invalidated.
- UNEXPECTED_STATE_E Spurious release failure.

C.47.2.56 function wc_RNG_lock_read

```
int wc_RNG_lock_read(
    WC_RNG * rng,
    WC_RNG_lock_arg_t * state
)
```

Read rng's lock word: the held/required latch bits, the entropy-invalidated bit, and any caller extra bits. The snapshot is racy by design.

Parameters:

- **rng** The RNG object to interrogate.
- **state** Receives the lock word.

See:

- wc_RNG_lock_get
- wc_RNG_invalidate_entropy

Return:

- 0 Success
- BAD_FUNC_ARG rng or state is null.

C.47.2.57 function wc_RNG_lock_set_extra

```
int wc_RNG_lock_set_extra(
    WC_RNG * rng,
    WC_RNG_lock_arg_t extra_bits
)
```

Atomically set (or) the supplied caller extra bits in rng's lock word. The caller should hold the latch.

Parameters:

- **rng** The RNG object to modify.
- **extra_bits** The bits to set.

See:

- wc_RNG_lock_add_extra
- wc_RNG_lock_clear_extra

Return:

- 0 Success
- BAD_FUNC_ARG rng is null.

C.47.2.58 function wc_RNG_lock_add_extra

```
int wc_RNG_lock_add_extra(
    WC_RNG * rng,
    WC_RNG_lock_arg_t extra_bits
)
```

Atomically add the supplied value to the caller extra-bits field of rng's lock word – for counters carried above WC_RNG_LOCK_EXTRA_SHIFT.

Parameters:

- **rng** The RNG object to modify.
- **extra_bits** The value to add.

See:

- wc_RNG_lock_set_extra
- wc_RNG_lock_clear_extra

Return:

- 0 Success
- BAD_FUNC_ARG rng is null.

C.47.2.59 function wc_RNG_lock_clear_extra

```
int wc_RNG_lock_clear_extra(
    WC_RNG * rng,
    WC_RNG_lock_arg_t extra_bits
)
```

Atomically clear the supplied caller extra bits in rng's lock word.

Parameters:

- **rng** The RNG object to modify.
- **extra_bits** The bits to clear.

See:

- wc_RNG_lock_set_extra
- wc_RNG_lock_add_extra

Return:

- 0 Success
- BAD_FUNC_ARG rng is null.

C.47.2.60 function wc_RNG_invalidate_entropy

```
int wc_RNG_invalidate_entropy(
    WC_RNG * rng
)
```

Mark rng's seed material untrusted – for VM fork/resume and similar duplication events – opening a new invalidation epoch: banked and pooled pre-event material is purged and wiped first, then the entropy-invalidated bit is latched in the lock word. An invalidated instance refuses service (NEEDS_RECOVERY_E) until recovery-reseeded. Latch or condemn: on any error return the latch is down, and the instance is instead condemned (status DRBG_FAILED). A condemned bank instance is retired and recovered by the entropy daemon; a condemned leaf gets no daemon rescue – its

owner sees RNG_FAILURE_E from subsequent operations and recovers it with wc_FreeRng() then wc_InitRng*().

Parameters:

- **rng** The RNG object to invalidate.

See:

- wc_RNG_lock_get
- wc_RNG_register_free_hook

Return:

- 0 Success: purges complete, latch asserted.
- BAD_FUNC_ARG rng is null.
- RNG_FAILURE_E (or other nonzero) A purge or the latch failed; the instance is condemned as above.

Example

```
// VM-resume handler
(void)wc_RNG_invalidate_entropy(rng);
// subsequent wc_RNG_lock_get() returns NEEDS_RECOVERY_E until recovery
```

C.47.2.61 function wc_RNG_register_free_hook

```
int wc_RNG_register_free_hook(
    WC_RNG * rng,
    wc_RNG_free_hook_cb_t free_hook,
    void * arg
)
```

Register a callback fired by wc_FreeRng() at teardown – for external registries (e.g. a kernel_module registry that must reach every live RNG on a VM duplication event) that need to drop their reference when the object dies.

Parameters:

- **rng** The RNG object to hook.
- **free_hook** The callback.
- **arg** Opaque argument passed to the callback.

See:

- wc_RNG_invalidate_entropy
- wc_FreeRng

Return:

- 0 Success
- BAD_FUNC_ARG rng or free_hook is null.

C.47.2.62 function wc_RNG_Pool_Alloc

```
int wc_RNG_Pool_Alloc(
    WC_RNG * rng,
    word32 size
)
```

Attach a random pool to rng: a buffer of size bytes of pre-generated output, filled by wc_RNG_Pool_Collect() and drained atomic_context_safely by wc_RNG_Pool_Extract(). The pool is released with the instance.

Parameters:

- **rng** The RNG object to equip.
- **size** Pool capacity in bytes.

See:

- wc_RNG_Pool_Collect
- wc_RNG_Pool_Extract

Return:

- 0 Success
- BAD_FUNC_ARG rng is null, or size is 0 or out of range.
- MEMORY_E Allocation failed.
- ALREADY_E The pool is already allocated.

Example

```
wc_RNG_Pool_Alloc(rng, 256);
wc_RNG_Pool_Collect(rng, 256);      // sleepable context
word32 n = 16;
if (wc_RNG_Pool_Extract(rng, out, &n) == 0) {
    // n bytes delivered, atomic-context-safe
}
```

C.47.2.63 function wc_RNG_Pool_Collect

```
int wc_RNG_Pool_Collect(
    WC_RNG * rng,
    word32 n
)
```

Generate up to n bytes into rng's pool from rng itself. The collect/extract hand-off is arbitrated by an atomic aperture word, so a single collector is safe alongside concurrent extractors.

Parameters:

- **rng** The RNG object whose pool to fill.
- **n** Maximum bytes to collect this call.

See:

- wc_RNG_Pool_Alloc
- wc_RNG_Pool_Collect2
- wc_RNG_Pool_Extract

Return:

- 0 Success
- ALREADY_E The pool is full or being drained.
- BAD_FUNC_ARG rng is null, has no pool, or n is 0.

C.47.2.64 function wc_RNG_Pool_Collect2

```
int wc_RNG_Pool_Collect2(
    WC_RNG * rng_dest,
    WC_RNG * rng_src,
    word32 n
)
```

The two-instance form of wc_RNG_Pool_Collect(): fill rng_dest's pool with output drawn from rng_src – so a service instance's pool can be topped up by a daemon-owned generator.

Parameters:

- **rng_dest** The RNG object whose pool to fill.
- **rng_src** The RNG object to draw output from.
- **n** Maximum bytes to collect this call.

See: wc_RNG_Pool_Collect

Return:

- 0 Success
- ALREADY_E The pool is full or being drained.
- BAD_FUNC_ARG rng_dest or rng_src is null, or rng_dest has no pool, or n is 0.
- BAD_STATE_E The pool is not allocated.
- NOT_READY_E The source could not serve; retry later.

C.47.2.65 function wc_RNG_Pool_Extract

```
int wc_RNG_Pool_Extract(
    WC_RNG * rng,
    byte * out,
    word32 * n
)
```

Drain up to *n bytes from rng's pool into out – atomic-context-safe. On success* n reports the bytes actually delivered; on any error return *n is left unmodified.

Parameters:

- **rng** The RNG object whose pool to drain.
- **out** Receives the output.
- **n** In: bytes requested; out: bytes delivered.

See:

- wc_RNG_Pool_Collect
- wc_RNG_Pool_Current

Return:

- 0 Success
- NOT_READY_E The pool is empty or being filled.
- BAD_FUNC_ARG rng, out, or n is null, or rng has no pool.
- BAD_STATE_E The pool is not allocated.
- RNG_FAILURE_E The instance is out of service.

C.47.2.66 function wc_RNG_Pool_Current

```
int wc_RNG_Pool_Current(
    WC_RNG * rng,
    word32 * n
)
```

Report the pool's current fill in bytes. The snapshot is racy by design.

Parameters:

- **rng** The RNG object to interrogate.
- **n** Receives the fill.

See: wc_RNG_Pool_Extract

Return:

- 0 Success
- BAD_FUNC_ARG rng or n is null, or rng has no pool.

C.47.2.67 function wc_rng_debug_stats_snap

```
int wc_rng_debug_stats_snap(
    struct wc_rng_debug_stats_snapshot * s,
    const WC_RNG * rng
)
```

Snapshot the global RNG debug counters (WC_RNG_DEBUG_STATS) – seeds and reseeds by provenance, generates, pool and bank traffic – into s, for later delta accounting with wc_rng_debug_stats_sum().

Parameters:

- **s** Receives the snapshot.
- **rng** Optional instance for per-instance context, or null.

See:

- wc_rng_debug_stats_sum
- wc_rng_debug_stats_restore

Return:

- 0 Success
- BAD_FUNC_ARG s is null.

C.47.2.68 function wc_rng_debug_stats_restore

```
int wc_rng_debug_stats_restore(
    const struct wc_rng_debug_stats_snapshot * s,
    WC_RNG * rng
)
```

Restore the global RNG debug counters from a snapshot – so a test can unwind its own accounting.

Parameters:

- **s** The snapshot to restore from.
- **rng** Optional instance for per-instance context, or null.

See: wc_rng_debug_stats_snap

Return:

- 0 Success
- BAD_FUNC_ARG s is null.

C.47.2.69 function wc_rng_debug_stats_sum

```
int wc_rng_debug_stats_sum(
    struct wc_rng_debug_stats_snapshot * s,
    const WC_RNG * rng
)
```

Accumulate the current global RNG debug counters into s – combined with a prior wc_rng_debug_stats_snap(), a delta accounting of the interval's RNG activity.

Parameters:

- **s** The snapshot to accumulate into.
- **rng** Optional instance for per-instance context, or null.

See: wc_rng_debug_stats_snap

Return:

- 0 Success
- BAD_FUNC_ARG s is null.

C.47.3 Source code

```
int  wc_InitNetRandom(const char* configFile, wnr_hmac_key hmac_cb, int
 ↪  timeout);

int  wc_FreeNetRandom(void);

int  wc_InitRng(WC_RNG* rng);

int  wc_RNG_GenerateBlock(WC_RNG* rng, byte* output, word32 sz);

int  wc_RNG_GenerateByte(WC_RNG* rng, byte* b);

int  wc_FreeRng(WC_RNG* rng);

int wc_RNG_HealthTest(int reseed, const byte* seedA, word32 seedASz,
        const byte* seedB, word32 seedBSz,
        byte* output, word32 outputSz);

int wc_GenerateSeed(OS_Seed* os, byte* output, word32 sz);

WC_RNG* wc_rng_new(byte* nonce, word32 nonceSz, void* heap);

int wc_rng_new_ex(WC_RNG **rng, byte* nonce, word32 nonceSz, void* heap,
                int devId);

void wc_rng_free(WC_RNG* rng);

int wc_InitRng_ex(WC_RNG* rng, void* heap, int devId);

int wc_InitRngNonce(WC_RNG* rng, const byte* nonce, word32 nonceSz);

int wc_InitRngNonce_ex(WC_RNG* rng, const byte* nonce, word32 nonceSz,
                     void* heap, int devId);

int wc_SetSeed_Cb(wc_RngSeed_Cb cb);

int wc_RNG_DRBG_Reseed(WC_RNG* rng, const byte* seed, word32 seedSz);

int wc_RNG_TestSeed(const byte* seed, word32 seedSz);
```

```
int wc_RNG_HealthTest_ex(int reseed, const byte* nonce, word32 nonceSz,
                         const byte* seedA, word32 seedASz,
                         const byte* seedB, word32 seedBSz, byte* output,
                         word32 outputSz, void* heap, int devId);

int wc_RNG_HealthTest_SHA512(int reseed, const byte* seedA, word32 seedASz,
        const byte* seedB, word32 seedBSz,
        byte* output, word32 outputSz);

int wc_RNG_HealthTest_SHA512_ex(int reseed, const byte* nonce, word32 nonceSz,
        const byte* persoString, word32 persoStringSz,
        const byte* seedA, word32 seedASz,
        const byte* seedB, word32 seedBSz,
        const byte* additionalA, word32 additionalASz,
        const byte* additionalB, word32 additionalBSz,
        byte* output, word32 outputSz,
        void* heap, int devId);

int wc_Sha256Drbg_Disable(void);

int wc_Sha256Drbg_Enable(void);

int wc_Sha256Drbg_IsDisabled(void);

int wc_Sha512Drbg_Disable(void);

int wc_Sha512Drbg_Enable(void);

int wc_Sha512Drbg_IsDisabled(void);

int wc_InitRng_ex2(WC_RNG* rng, void* heap, int devId, word32 flags);

int wc_InitRngNonce_ex2(WC_RNG* rng, const byte* nonce, word32 nonceSz,
                        const byte *perso, word32 persoSz,
                        void* heap, int devId, word32 flags);

int wc_RNG_GetStatus(const WC_RNG* rng);

int wc_RNG_DRBG_Present(const WC_RNG* rng);

int wc_RNG_DRBG_GetReseedCtr(const WC_RNG* rng, wc_drbg_reseed_ctr_t*
 ↪  reseedCtr);

int wc_RNG_DRBG_ScheduleReseed(WC_RNG* rng);

int wc_RNG_DRBG_Reseed_Now(WC_RNG* rng, const byte* nonce, word32 nonceSz);

int wc_RNG_DRBG_Reseed_Nonce(WC_RNG* rng, const byte* seed, word32 seedSz,
                             const byte *nonce, word32 nonceSz);

int wc_RNG_DRBG_Stir(WC_RNG* rng, const byte* seed, word32 seedSz);

int wc_RNG_DRBG_Stir_Nonce(WC_RNG* rng, const byte* seed,
```

```
                                        word32 seedSz, const byte *nonce,
                                        word32 nonceSz);

int wc_InitRngRBGC(WC_RNG* child, WC_RNG* parent, word32 flags);

int wc_InitRngNonceRBGC(WC_RNG* child, WC_RNG* parent, const byte* nonce,
                        word32 nonceSz, word32 flags);

int wc_InitRngRBGC_New(WC_RNG** child, WC_RNG* parent, word32 flags);

int wc_InitRngNonceRBGC_New(WC_RNG** child, WC_RNG* parent, const byte* nonce,
                            word32 nonceSz, word32 flags);

int wc_RNG_DRBG_ReseedRBGC(WC_RNG* rng, WC_RNG* root, const byte* nonce,
                           word32 nonceSz);

int wc_RNG_DRBG_StirRBGC(WC_RNG* rng, WC_RNG* root,
                                       const byte* nonce, word32 nonceSz);

int wc_RNG_DRBG_GetRBGCStratum(const WC_RNG* rng);

int wc_RNG_DRBG_GetNextSeedRBGCStratum(const WC_RNG* rng);

int wc_RNG_DRBG_NextSeedGenerate(WC_RNG* rng, word32 n);

int wc_RNG_DRBG_NextSeedGenerate_RBGC(WC_RNG* rng, WC_RNG *root, word32 n);

int wc_RNG_DRBG_NextSeedCurrent(WC_RNG* rng, WC_ATOMIC_INT_ARG* n);

int wc_RNG_DRBG_NextSeedNow(WC_RNG* rng);

int wc_RNG_DRBG_NextSeedNow_Nonce(WC_RNG* rng, const byte* nonce,
                                  word32 nonceSz);

int wc_RNG_DRBG_NextStirStore(WC_RNG* rng, const byte *nonce,
                                       word32 nonceSz);

int wc_RNG_DRBG_NextStirNow(WC_RNG* rng);

int wc_RNG_lock_get(WC_RNG* rng, WC_RNG_lock_arg_t extra_bits);

int wc_RNG_lock_get_conditional(WC_RNG* rng,
                                WC_RNG_lock_arg_t expected_extra_bits,
                                WC_RNG_lock_arg_t want_extra_bits);

int wc_RNG_lock_put(WC_RNG* rng, WC_RNG_lock_arg_t extra_bits);

int wc_RNG_lock_put_conditional(WC_RNG* rng,
                                WC_RNG_lock_arg_t expected_extra_bits,
                                WC_RNG_lock_arg_t want_extra_bits);

int wc_RNG_lock_read(WC_RNG* rng, WC_RNG_lock_arg_t* state);

int wc_RNG_lock_set_extra(WC_RNG* rng, WC_RNG_lock_arg_t extra_bits);
```

```
int wc_RNG_lock_add_extra(WC_RNG* rng, WC_RNG_lock_arg_t extra_bits);

int wc_RNG_lock_clear_extra(WC_RNG* rng, WC_RNG_lock_arg_t extra_bits);

int wc_RNG_invalidate_entropy(WC_RNG* rng);

int wc_RNG_register_free_hook(WC_RNG* rng, wc_RNG_free_hook_cb_t free_hook,
                              void *arg);

int wc_RNG_Pool_Alloc(WC_RNG* rng, word32 size);

int wc_RNG_Pool_Collect(WC_RNG* rng, word32 n);

int wc_RNG_Pool_Collect2(WC_RNG* rng_dest, WC_RNG* rng_src, word32 n);

int wc_RNG_Pool_Extract(WC_RNG* rng, byte* out, word32* n);

int wc_RNG_Pool_Current(WC_RNG* rng, word32* n);

int wc_rng_debug_stats_snap(struct wc_rng_debug_stats_snapshot *s,
                            const WC_RNG *rng);

int wc_rng_debug_stats_restore(const struct wc_rng_debug_stats_snapshot *s,
                               WC_RNG *rng);

int wc_rng_debug_stats_sum(struct wc_rng_debug_stats_snapshot *s,
                           const WC_RNG *rng);
```

C.48 ripemd.h

C.48.1 Functions

	Name
int	**wc_InitRipeMd**(RipeMd * ripemd)This function initializes a ripemd structure by initializing ripemd's digest, buffer, loLen and hiLen.
int	**wc_RipeMdUpdate**(RipeMd * ripemd, const byte * data, word32 len)This function generates the RipeMd digest of the data input and stores the result in the ripemd->digest buffer. After running wc_RipeMdUpdate, one should compare the generated ripemd->digest to a known authentication tag to verify the authenticity of a message.
int	**wc_RipeMdFinal**(RipeMd * ripemd, byte * hash)This function copies the computed digest into hash. If there is a partial unhashed block, this method will pad the block with 0s, and include that block's round in the digest before copying to hash. State of ripemd is reset.

C.48.2 Functions Documentation

C.48.2.1 function wc_InitRipeMd

```
int wc_InitRipeMd(
    RipeMd * ripemd
)
```

This function initializes a ripemd structure by initializing ripemd's digest, buffer, loLen and hiLen.

Parameters:

- **ripemd** pointer to the ripemd structure to initialize

See:

- wc_RipeMdUpdate
- wc_RipeMdFinal

Return:

- 0 returned on successful execution of the function. The RipeMd structure is initialized.
- BAD_FUNC_ARG returned if the RipeMd structure is NULL.

Example

```
RipeMd md;
int ret;
ret = wc_InitRipeMd(&md);
if (ret != 0) {
    // Failure case.
}
```

C.48.2.2 function wc_RipeMdUpdate

```
int wc_RipeMdUpdate(
    RipeMd * ripemd,
    const byte * data,
    word32 len
)
```

This function generates the RipeMd digest of the data input and stores the result in the ripemd->digest buffer. After running wc_RipeMdUpdate, one should compare the generated ripemd->digest to a known authentication tag to verify the authenticity of a message.

Parameters:

- **ripemd** pointer to the ripemd structure to be initialized with wc_InitRipeMd
- **data** data to be hashed
- **len** sizeof data in bytes

See:

- wc_InitRipeMd
- wc_RipeMdFinal

Return:

- 0 Returned on successful execution of the function.
- BAD_FUNC_ARG Returned if the RipeMd structure is NULL or if data is NULL and len is not zero. This function should execute if data is NULL and len is 0.

Example

```
const byte* data; // The data to be hashed
....
RipeMd md;
int ret;
ret = wc_InitRipeMd(&md);
if (ret == 0) {
ret = wc_RipeMdUpdate(&md, plain, sizeof(plain));
if (ret != 0) {
// Failure case …
```

C.48.2.3 function wc_RipeMdFinal

```
int wc_RipeMdFinal(
    RipeMd * ripemd,
    byte * hash
)
```

This function copies the computed digest into hash. If there is a partial unhashed block, this method will pad the block with 0s, and include that block's round in the digest before copying to hash. State of ripemd is reset.

Parameters:

- **ripemd** pointer to the ripemd structure to be initialized with wc_InitRipeMd, and containing hashes from wc_RipeMdUpdate. State will be reset
- **hash** buffer to copy digest to. Should be RIPEMD_DIGEST_SIZE bytes

See: none

Return:

- 0 Returned on successful execution of the function. The state of the RipeMd structure has been reset.
- BAD_FUNC_ARG Returned if the RipeMd structure or hash parameters are NULL.

Example

```
RipeMd md;
int ret;
byte   digest[RIPEMD_DIGEST_SIZE];
const byte* data; // The data to be hashed
...
ret = wc_InitRipeMd(&md);
if (ret == 0) {
ret = wc_RipeMdUpdate(&md, plain, sizeof(plain));
    if (ret != 0) {
        // RipeMd Update Failure Case.
}
ret = wc_RipeMdFinal(&md, digest);
if (ret != 0) {
    // RipeMd Final Failure Case.
}...
```

C.48.3 Source code

```
int wc_InitRipeMd(RipeMd* ripemd);
```

```
int wc_RipeMdUpdate(RipeMd* ripemd, const byte* data, word32 len);

int wc_RipeMdFinal(RipeMd* ripemd, byte* hash);
```

C.49 rsa.h

C.49.1 Functions

	Name
int	**wc_InitRsaKey**(RsaKey * key, void * heap)This function initializes a provided RsaKey struct. It also takes in a heap identifier, for use with user defined memory overrides (see XMALLOC, XFREE, XREALLOC).
int	**wc_InitRsaKey_Id**(RsaKey * key, unsigned char * id, int len, void * heap, int devId)This function initializes a provided RsaKey struct. The id and len are used to identify the key on the device while the devId identifies the device. It also takes in a heap identifier, for use with user defined memory overrides (see XMALLOC, XFREE, XREALLOC).
int	**wc_RsaSetRNG**(RsaKey * key, WC_RNG * rng)This function associates RNG with Key. It is needed when WC_RSA_BLINDING is enabled.
int	**wc_FreeRsaKey**(RsaKey * key)This function frees a provided RsaKey struct using mp_clear.
int	**wc_RsaDirect**(const byte * in, word32 inLen, byte * out, word32 * outSz, RsaKey * key, int type, WC_RNG * rng)Function that does the RSA operation directly with no padding. The input size must match key size. Typically this is used when padding is already done on the RSA input.
int	**wc_RsaPublicEncrypt**(const byte * in, word32 inLen, byte * out, word32 outLen, RsaKey * key, WC_RNG * rng)This function encrypts a message from in and stores the result in out. It requires an initialized public key and a random number generator. As a side effect, this function will return the bytes written to out in outLen.
int	**wc_RsaPrivateDecryptInline**(byte * in, word32 inLen, byte ** out, RsaKey * key)This functions is utilized by the wc_RsaPrivateDecrypt function for decrypting.
int	**wc_RsaPrivateDecrypt**(const byte * in, word32 inLen, byte * out, word32 outLen, RsaKey * key)This functions provides private RSA decryption.

	Name
int	**wc_RsaSSL_Sign**(const byte * in, word32 inLen, byte * out, word32 outLen, RsaKey * key, WC_RNG * rng)Signs the provided array with the private key.
int	**wc_RsaSSL_VerifyInline**(byte * in, word32 inLen, byte ** out, RsaKey * key)Used to verify that the message was signed by RSA key. The output uses the same byte array as the input.
int	**wc_RsaSSL_Verify**(const byte * in, word32 inLen, byte * out, word32 outLen, RsaKey * key)Used to verify that the message was signed by key.
int	**wc_RsaPSS_Sign**(const byte * in, word32 inLen, byte * out, word32 outLen, enum wc_HashType hash, int mgf, RsaKey * key, WC_RNG * rng)Signs the provided array with the private key.
int	**wc_RsaPSS_Verify**(const byte * in, word32 inLen, byte * out, word32 outLen, enum wc_HashType hash, int mgf, RsaKey * key)Decrypt input signature to verify that the message was signed by key. The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.
int	**wc_RsaPSS_VerifyInline**(byte * in, word32 inLen, byte ** out, enum wc_HashType hash, int mgf, RsaKey * key)Decrypt input signature to verify that the message was signed by RSA key. The output uses the same byte array as the input. The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.
int	**wc_RsaPSS_VerifyCheck**(const byte * in, word32 inLen, byte * out, word32 outLen, const byte * digest, word32 digestLen, enum wc_HashType hash, int mgf, RsaKey * key)Verify the message signed with RSA-PSS. Salt length is equal to hash length. The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.
int	**wc_RsaPSS_VerifyCheck_ex**(byte * in, word32 inLen, byte * out, word32 outLen, const byte * digest, word32 digestLen, enum wc_HashType hash, int mgf, int saltLen, RsaKey * key)Verify the message signed with RSA-PSS. The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.

	Name
int	**wc_RsaPSS_VerifyCheckInline**(byte * in, word32 inLen, byte ** out, const byte * digest, word32 digentLen, enum wc_HashType hash, int mgf, RsaKey * key)Verify the message signed with RSA-PSS. The input buffer is reused for the output buffer. Salt length is equal to hash length.
int	**wc_RsaPSS_VerifyCheckInline_ex**(byte * in, word32 inLen, byte ** out, const byte * digest, word32 digentLen, enum wc_HashType hash, int mgf, int saltLen, RsaKey * key)Verify the message signed with RSA-PSS. The input buffer is reused for the output buffer. The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.
int	**wc_RsaPSS_CheckPadding**(const byte * in, word32 inLen, const byte * sig, word32 sigSz, enum wc_HashType hashType)Checks the PSS data to ensure that the signature matches. Salt length is equal to hash length. The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.
int	**wc_RsaPSS_CheckPadding_ex**(const byte * in, word32 inLen, const byte * sig, word32 sigSz, enum wc_HashType hashType, int saltLen, int bits)Checks the PSS data to ensure that the signature matches. Salt length is equal to hash length.
int	**wc_RsaEncryptSize**(const RsaKey * key)Returns the encryption size for the provided key structure.
int	**wc_RsaPrivateKeyDecode**(const byte * input, word32 * inOutIdx, RsaKey * key, word32 inSz)This function parses a DER-formatted RSA private key, extracts the private key and stores it in the given RsaKey structure. It also sets the distance parsed in idx.
int	**wc_RsaPublicKeyDecode**(const byte * input, word32 * inOutIdx, RsaKey * key, word32 inSz)This function parses a DER-formatted RSA public key, extracts the public key and stores it in the given RsaKey structure. It also sets the distance parsed in idx.
int	**wc_RsaPublicKeyDecodeRaw**(const byte * n, word32 nSz, const byte * e, word32 eSz, RsaKey * key)This function decodes the raw elements of an RSA public key, taking in the public modulus (n) and exponent (e). It stores these raw elements in the provided RsaKey structure, allowing one to use them in the encryption/decryption process.

	Name
int	**wc_RsaKeyToDer**(RsaKey * key, byte * output, word32 outLen)This function converts an RsaKey key to DER format. The result is written to output and it returns the number of bytes written.
int	**wc_RsaPublicEncrypt_ex**(const byte * in, word32 inLen, byte * out, word32 outLen, RsaKey * key, WC_RNG * rng, int type, enum wc_HashType hash, int mgf, byte * label, word32 labelSz)This function performs RSA encrypt while allowing the choice of which padding to use.
int	**wc_RsaPrivateDecrypt_ex**(const byte * in, word32 inLen, byte * out, word32 outLen, RsaKey * key, int type, enum wc_HashType hash, int mgf, byte * label, word32 labelSz)This function uses RSA to decrypt a message and gives the option of what padding type.
int	**wc_RsaPrivateDecryptInline_ex**(byte * in, word32 inLen, byte ** out, RsaKey * key, int type, enum wc_HashType hash, int mgf, byte * label, word32 labelSz)This function uses RSA to decrypt a message inline and gives the option of what padding type. The in buffer will contain the decrypted message after being called and the out byte pointer will point to the location in the “in” buffer where the plain text is.
int	**wc_RsaFlattenPublicKey**(const RsaKey * key, byte * e, word32 * eSz, byte * n, word32 * nSz)Flattens the RsaKey structure into individual elements (e, n) used for the RSA algorithm.
int	**wc_RsaKeyToPublicDer**(RsaKey * key, byte * output, word32 outLen)Convert Rsa Public key to DER format. Writes to output, and returns count of bytes written.
int	**wc_RsaKeyToPublicDer_ex**(RsaKey * key, byte * output, word32 outLen, int with_header)Convert RSA Public key to DER format. Writes to output, and returns count of bytes written. If with_header is 0 then only the (seq + n + e) is returned in ASN.1 DER format and will exclude the header.

	Name
int	**wc_MakeRsaKey**(RsaKey * key, int size, long e, WC_RNG * rng)This function generates a RSA private key of length size (in bits) and given exponent (e). It then stores this key in the provided RsaKey structure, so that it may be used for encryption/decryption. A secure number to use for e is 65537. size is required to be greater than or equal to RSA_MIN_SIZE and less than or equal to RSA_MAX_SIZE. For this function to be available, the option WOLFSSL_KEY_GEN must be enabled at compile time. This can be accomplished with –enable-keygen if using ./configure.
int	**wc_RsaSetNonBlock**(RsaKey * key, RsaNb * nb)This function sets the non-blocking RSA context. When a RsaNb context is set it enables fast math based non-blocking exptmod, which splits the RSA function into many smaller operations. Enabled when WC_RSA_NONBLOCK is defined.
int	**wc_RsaSetNonBlockTime**(RsaKey * key, word32 maxBlockUs, word32 cpuMHz)This function configures the maximum amount of blocking time in microseconds. It uses a pre_computed table (see tfm.c exptModNbInst) along with the CPU speed in megahertz to determine if the next operation can be completed within the maximum blocking time provided. Enabled when WC_RSA_NONBLOCK_TIME is defined.
int	**wc_InitRsaKey_ex**(RsaKey * key, void * heap, int devId)Initializes RSA key with heap and device ID.
RsaKey *	**wc_NewRsaKey**(void * heap, int devId, int * result_code)Allocates and initializes new RSA key. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.
int	**wc_DeleteRsaKey**(RsaKey * key, RsaKey ** key_p)Deletes and frees RSA key. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.
int	**wc_InitRsaKey_Label**(RsaKey * key, const char * label, void * heap, int devId)Initializes RSA key with label.
int	**wc_CheckRsaKey**(RsaKey * key)Checks RSA key validity.
int	**wc_RsaUseKeyId**(RsaKey * key, word32 keyId, word32 flags)Uses key ID for hardware RSA.

	Name
int	**wc_RsaGetKeyId**(RsaKey * key, word32 * keyId)Gets key ID from hardware RSA key.
int	**wc_RsaFunction**(const byte * in, word32 inLen, byte * out, word32 * outLen, int type, RsaKey * key, WC_RNG * rng)Performs RSA operation.
int	**wc_RsaPSS_Sign_ex**(const byte * in, word32 inLen, byte * out, word32 outLen, enum wc_HashType hash, int mgf, int saltLen, RsaKey * key, WC_RNG * rng)Signs with RSA-PSS extended options.
int	**wc_RsaSSL_Verify_ex**(const byte * in, word32 inLen, byte * out, word32 outLen, RsaKey * key, int pad_type)Verifies RSA signature with padding type.
int	**wc_RsaSSL_Verify_ex2**(const byte * in, word32 inLen, byte * out, word32 outLen, RsaKey * key, int pad_type, enum wc_HashType hash)Verifies RSA signature with hash type.
int	**wc_RsaPSS_VerifyInline_ex**(byte * in, word32 inLen, byte ** out, enum wc_HashType hash, int mgf, int saltLen, RsaKey * key)Verifies RSA-PSS inline with extended options.
int	**wc_RsaPSS_Verify_ex**(const byte * in, word32 inLen, byte * out, word32 outLen, enum wc_HashType hash, int mgf, int saltLen, RsaKey * key)Verifies RSA-PSS with extended options.
int	**wc_RsaPSS_CheckPadding_ex2**(const byte * in, word32 inLen, const byte * sig, word32 sigSz, enum wc_HashType hashType, int saltLen, int bits, void * heap)Checks RSA-PSS padding with extended options.
int	**wc_RsaExportKey**(const RsaKey * key, byte * e, word32 * eSz, byte * n, word32 * nSz, byte * d, word32 * dSz, byte * p, word32 * pSz, byte * q, word32 * qSz)Exports RSA key components.
int	**wc_CheckProbablePrime_ex**(const byte * p, word32 pSz, const byte * q, word32 qSz, const byte * e, word32 eSz, int nlen, int * isPrime, WC_RNG * rng)Checks probable prime with extended options.
int	**wc_CheckProbablePrime**(const byte * p, word32 pSz, const byte * q, word32 qSz, const byte * e, word32 eSz, int nlen, int * isPrime)Checks probable prime.
int	**wc_RsaPad_ex**(const byte * input, word32 inputLen, byte * pkcsBlock, word32 pkcsBlockLen, byte padValue, WC_RNG * rng, int padType, enum wc_HashType hType, int mgf, byte * optLabel, word32 labelLen, int saltLen, int bits, void * heap)Pads data with extended options.

COPYRIGHT ©2024 wolfSSL Inc.

	Name
int	**wc_RsaUnPad_ex**(byte * pkcsBlock, word32 pkcsBlockLen, byte ** out, byte padValue, int padType, enum wc_HashType hType, int mgf, byte * optLabel, word32 labelLen, int saltLen, int bits, void * heap)Unpads data with extended options.
int	**wc_RsaPrivateKeyDecodeRaw**(const byte * n, word32 nSz, const byte * e, word32 eSz, const byte * d, word32 dSz, const byte * u, word32 uSz, const byte * p, word32 pSz, const byte * q, word32 qSz, const byte * dP, word32 dPSz, const byte * dQ, word32 dQSz, RsaKey * key)Decodes raw RSA private key.

C.49.2 Functions Documentation

C.49.2.1 function wc_InitRsaKey

```
int wc_InitRsaKey(
    RsaKey * key,
    void * heap
)
```

This function initializes a provided RsaKey struct. It also takes in a heap identifier, for use with user defined memory overrides (see XMALLOC, XFREE, XREALLOC).

Parameters:

- **key** pointer to the RsaKey structure to initialize
- **heap** pointer to a heap identifier, for use with memory overrides, allowing custom handling of memory allocation. This heap will be the default used when allocating memory for use with this RSA object

See:

- wc_FreeRsaKey
- wc_RsaSetRNG

Return:

- 0 Returned upon successfully initializing the RSA structure for use with encryption and decryption
- BAD_FUNC_ARGS Returned if the RSA key pointer evaluates to NULL

The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.

Example

```
RsaKey enc;
int ret;
ret = wc_InitRsaKey(&enc, NULL); // not using heap hint. No custom memory
if ( ret != 0 ) {
    // error initializing RSA key
}
```

C.49.2.2 function wc_InitRsaKey_Id

```
int wc_InitRsaKey_Id(
    RsaKey * key,
    unsigned char * id,
    int len,
    void * heap,
    int devId
)
```

This function initializes a provided RsaKey struct. The id and len are used to identify the key on the device while the devId identifies the device. It also takes in a heap identifier, for use with user defined memory overrides (see XMALLOC, XFREE, XREALLOC).

Parameters:

- **key** pointer to the RsaKey structure to initialize
- **id** identifier of key on device
- **len** length of identifier in bytes
- **heap** pointer to a heap identifier, for use with memory overrides, allowing custom handling of memory allocation. This heap will be the default used when allocating memory for use with this RSA object
- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID (-2) if not used

See:

- wc_InitRsaKey
- wc_FreeRsaKey
- wc_RsaSetRNG

Return:

- 0 Returned upon successfully initializing the RSA structure for use with encryption and decryption
- BAD_FUNC_ARGS Returned if the RSA key pointer evaluates to NULL, or if id is NULL while len is positive
- BUFFER_E Returned if len is less than 0 or greater than RSA_MAX_ID_LEN.

Note: This API is only available when WOLF_PRIVATE_KEY_ID is defined, which is set for PKCS11 support.

The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.

Example

```
RsaKey enc;
unsigned char* id = (unsigned char*)"RSA2048";
int len = 7;
int devId = 1;
int ret;
ret = wc_CryptoDev_RegisterDevice(devId, wc_Pkcs11_CryptoDevCb,
                                  &token);
if ( ret != 0) {
    // error associating callback and token with device id
}
ret = wc_InitRsaKey_Id(&enc, id, len, NULL, devId); // not using heap hint
if ( ret != 0 ) {
    // error initializing RSA key
}
```

C.49.2.3 function wc_RsaSetRNG

```
int wc_RsaSetRNG(
    RsaKey * key,
    WC_RNG * rng
)
```

This function associates RNG with Key. It is needed when WC_RSA_BLINDING is enabled.

Parameters:

- **key** pointer to the RsaKey structure to be associated
- **rng** pointer to the WC_RNG structure to associate with

See:

- wc_InitRsaKey
- wc_RsaSetRNG

Return:

- 0 Returned upon success
- BAD_FUNC_ARGS Returned if the RSA key, rng pointer evaluates to NULL

Example

```
ret = wc_InitRsaKey(&key, NULL);
if (ret == 0) {
    ret = wc_InitRng(&rng);
} else return -1;
if (ret == 0) {
    ret = wc_RsaSetRNG(&key, &rng);
```

C.49.2.4 function wc_FreeRsaKey

```
int wc_FreeRsaKey(
    RsaKey * key
)
```

This function frees a provided RsaKey struct using mp_clear.

Parameters:

- **key** pointer to the RsaKey structure to free

See: wc_InitRsaKey

Return: 0 Returned upon successfully freeing the key

Example

```
RsaKey enc;
wc_InitRsaKey(&enc, NULL); // not using heap hint. No custom memory
... set key, do encryption

wc_FreeRsaKey(&enc);
```

C.49.2.5 function wc_RsaDirect

```
int wc_RsaDirect(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outSz,
```

```
    RsaKey * key,
    int type,
    WC_RNG * rng
)
```

Function that does the RSA operation directly with no padding. The input size must match key size. Typically this is used when padding is already done on the RSA input.

Parameters:

- **in** buffer to do operation on
- **inLen** length of input buffer
- **out** buffer to hold results
- **outSz** gets set to size of result buffer. Should be passed in as length of out buffer. If the pointer "out" is null then outSz gets set to the expected buffer size needed and LENGTH_ONLY_E gets returned.
- **key** initialized RSA key to use for encrypt/decrypt
- **type** if using private or public key (RSA_PUBLIC_ENCRYPT, RSA_PUBLIC_DECRYPT, RSA_PRIVATE_ENCRYPT, RSA_PRIVATE_DECRYPT)
- **rng** initialized WC_RNG struct

See:

- wc_RsaPublicEncrypt
- wc_RsaPrivateDecrypt

Return:

- size On successfully encryption the size of the encrypted buffer is returned
- RSA_BUFFER_E RSA buffer error, output too small or input too large

Example

```
int ret;
WC_RNG rng;
RsaKey key;
byte  in[256];
byte out[256];
word32 outSz = (word32)sizeof(out);
…

ret = wc_RsaDirect(in, (word32)sizeof(in), out, &outSz, &key,
    RSA_PRIVATE_ENCRYPT, &rng);
if (ret < 0) {
    //handle error
}
```

C.49.2.6 function wc_RsaPublicEncrypt

```
int wc_RsaPublicEncrypt(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    RsaKey * key,
    WC_RNG * rng
)
```

This function encrypts a message from in and stores the result in out. It requires an initialized public key and a random number generator. As a side effect, this function will return the bytes written to out in outLen.

Parameters:

- **in** pointer to a buffer containing the input message to encrypt
- **inLen** the length of the message to encrypt
- **out** pointer to the buffer in which to store the output ciphertext
- **outLen** the length of the output buffer
- **key** pointer to the RsaKey structure containing the public key to use for encryption
- **rng** The RNG structure with which to generate random block padding

See: wc_RsaPrivateDecrypt

Return:

- Success Upon successfully encrypting the input message, returns the number of bytes written on success and less than zero for failure.
- BAD_FUNC_ARG Returned if any of the input parameters are invalid
- RSA_BUFFER_E Returned if the output buffer is too small to store the ciphertext
- RNG_FAILURE_E Returned if there is an error generating a random block using the provided RNG structure
- MP_INIT_E May be returned if there is an error in the math library used while encrypting the message
- MP_READ_E May be returned if there is an error in the math library used while encrypting the message
- MP_CMP_E May be returned if there is an error in the math library used while encrypting the message
- MP_INVMOD_E May be returned if there is an error in the math library used while encrypting the message
- MP_EXPTMOD_E May be returned if there is an error in the math library used while encrypting the message
- MP_MOD_E May be returned if there is an error in the math library used while encrypting the message
- MP_MUL_E May be returned if there is an error in the math library used while encrypting the message
- MP_ADD_E May be returned if there is an error in the math library used while encrypting the message
- MP_MULMOD_E May be returned if there is an error in the math library used while encrypting the message
- MP_TO_E May be returned if there is an error in the math library used while encrypting the message
- MP_MEM May be returned if there is an error in the math library used while encrypting the message
- MP_ZERO_E May be returned if there is an error in the math library used while encrypting the message

Example

```
RsaKey pub;
int ret = 0;
byte n[] = { // initialize with received n component of public key };
byte e[] = { // initialize with received e component of public key };
byte msg[] = { // initialize with plaintext of message to encrypt };
byte cipher[256]; // 256 bytes is large enough to store 2048 bit RSA
ciphertext
```

```
wc_InitRsaKey(&pub, NULL); // not using heap hint. No custom memory
wc_RsaPublicKeyDecodeRaw(n, sizeof(n), e, sizeof(e), &pub);
// initialize with received public key parameters
ret = wc_RsaPublicEncrypt(msg, sizeof(msg), out, sizeof(out), &pub, &rng);
if ( ret != 0 ) {
    // error encrypting message
}
```

C.49.2.7 function wc_RsaPrivateDecryptInline

```
int wc_RsaPrivateDecryptInline(
    byte * in,
    word32 inLen,
    byte ** out,
    RsaKey * key
)
```

This functions is utilized by the wc_RsaPrivateDecrypt function for decrypting.

Parameters:

- **in** The byte array to be decrypted.
- **inLen** The length of in.
- **out** The byte array for the decrypted data to be stored.
- **key** The key to use for decryption.

See: wc_RsaPrivateDecrypt

Return:

- Success Length of decrypted data.
- RSA_PAD_E RsaUnPad error, bad formatting

Example

none

C.49.2.8 function wc_RsaPrivateDecrypt

```
int wc_RsaPrivateDecrypt(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    RsaKey * key
)
```

This functions provides private RSA decryption.

Parameters:

- **in** The byte array to be decrypted.
- **inLen** The length of in.
- **out** The byte array for the decrypted data to be stored.
- **outLen** The length of out.
- **key** The key to use for decryption.

See:

- RsaUnPad
- wc_RsaFunction

- wc_RsaPrivateDecryptInline

Return:

- Success length of decrypted data.
- MEMORY_E -125, out of memory error
- BAD_FUNC_ARG -173, Bad function argument provided

Example

```
ret = wc_RsaPublicEncrypt(in, inLen, out, sizeof(out), &key, &rng);
if (ret < 0) {
    return -1;
}
ret = wc_RsaPrivateDecrypt(out, ret, plain, sizeof(plain), &key);
if (ret < 0) {
    return -1;
}
```

C.49.2.9 function wc_RsaSSL_Sign

```
int wc_RsaSSL_Sign(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    RsaKey * key,
    WC_RNG * rng
)
```

Signs the provided array with the private key.

Parameters:

- **in** The byte array to be encrypted.
- **inLen** The length of in.
- **out** The byte array for the encrypted data to be stored.
- **outLen** The length of out.
- **key** The key to use for encryption.
- **RNG** The RNG struct to use for random number purposes.

See: wc_RsaPad

Return: RSA_BUFFER_E: -131, RSA buffer error, output too small or input too large

Example

```
ret = wc_RsaSSL_Sign(in, inLen, out, sizeof(out), &key, &rng);
if (ret < 0) {
    return -1;
}
memset(plain, 0, sizeof(plain));
ret = wc_RsaSSL_Verify(out, ret, plain, sizeof(plain), &key);
if (ret < 0) {
    return -1;
}
if (ret != inLen) {
    return -1;
}
if (XMEMCMP(in, plain, ret) != 0) {
```

```
    return -1;
}
```

C.49.2.10 function wc_RsaSSL_VerifyInline

```
int wc_RsaSSL_VerifyInline(
    byte * in,
    word32 inLen,
    byte ** out,
    RsaKey * key
)
```

Used to verify that the message was signed by RSA key. The output uses the same byte array as the input.

Parameters:

- **in** Byte array to be decrypted.
- **inLen** Length of the buffer input.
- **out** Pointer to a pointer for decrypted information.
- **key** RsaKey to use.

See:

- wc_RsaSSL_Verify
- wc_RsaSSL_Sign

Return:

- 0 Length of the digest.
- <0 An error occurred.

Example

```
RsaKey key;
WC_RNG rng;
int ret = 0;
long e = 65537; // standard value to use for exponent
wc_InitRsaKey(&key, NULL); // not using heap hint. No custom memory
wc_InitRng(&rng);
wc_MakeRsaKey(&key, 2048, e, &rng);

byte in[] = { // Initialize with some RSA encrypted information }
byte* out;
if(wc_RsaSSL_VerifyInline(in, sizeof(in), &out, &key) < 0)
{
    // handle error
}
```

C.49.2.11 function wc_RsaSSL_Verify

```
int wc_RsaSSL_Verify(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    RsaKey * key
)
```

Used to verify that the message was signed by key.

Parameters:

- **in** The byte array to be decrypted.
- **inLen** The length of in.
- **out** The byte array for the decrypted data to be stored.
- **outLen** The length of out.
- **key** The key to use for verification.

See: wc_RsaSSL_Sign

Return:

- Success Length of digest on no error.
- MEMORY_E memory exception.

Example

```
ret = wc_RsaSSL_Sign(in, inLen, out, sizeof(out), &key, &rng);
if (ret < 0) {
    return -1;
}
memset(plain, 0, sizeof(plain));
ret = wc_RsaSSL_Verify(out, ret, plain, sizeof(plain), &key);
if (ret < 0) {
    return -1;
}
if (ret != inLen) {
    return -1;
}
if (XMEMCMP(in, plain, ret) != 0) {
    return -1;
}
```

C.49.2.12 function wc_RsaPSS_Sign

```
int wc_RsaPSS_Sign(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    enum wc_HashType hash,
    int mgf,
    RsaKey * key,
    WC_RNG * rng
)
```

Signs the provided array with the private key.

Parameters:

- **in** The byte array to be encrypted.
- **inLen** The length of in.
- **out** The byte array for the encrypted data to be stored.
- **outLen** The length of out.
- **hash** The hash type to be in message
- **mgf** Mask Generation Function Identifiers
- **key** The key to use for verification.

See:

- wc_RsaPSS_Verify
- wc_RsaSetRNG

Return: RSA_BUFFER_E: -131, RSA buffer error, output too small or input too large

Example

```
ret = wc_InitRsaKey(&key, NULL);
if (ret == 0) {
    ret = wc_InitRng(&rng);
} else return -1;
if (ret == 0) {
    ret = wc_RsaSetRNG(&key, &rng);
} else return -1;
if (ret == 0) {
        ret = wc_MakeRsaKey(&key, 2048, WC_RSA_EXPONENT, &rng);
} else return -1;

ret = wc_RsaPSS_Sign((byte*)szMessage, (word32)XSTRLEN(szMessage)+1,
        pSignature, sizeof(pSignature),
        WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key, &rng);
if (ret > 0 ){
    sz = ret;
} else return -1;

ret = wc_RsaPSS_Verify(pSignature, sz, pt, outLen,
        WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key);
if (ret <= 0)return -1;

wc_FreeRsaKey(&key);
wc_FreeRng(&rng);
```

C.49.2.13 function wc_RsaPSS_Verify

```
int wc_RsaPSS_Verify(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    enum wc_HashType hash,
    int mgf,
    RsaKey * key
)
```

Decrypt input signature to verify that the message was signed by key. The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.

Parameters:

- **in** The byte array to be decrypted.
- **inLen** The length of in.
- **out** The byte array for the decrypted data to be stored.
- **outLen** The length of out.
- **hash** The hash type to be in message
- **mgf** Mask Generation Function Identifiers
- **key** The key to use for verification.

See:

- wc_RsaPSS_Sign
- wc_RsaPSS_VerifyInline
- wc_RsaPSS_CheckPadding
- wc_RsaSetRNG

Return:

- Success Length of text on no error.
- MEMORY_E memory exception.
- MP_EXPTMOD_E - When using fastmath and FP_MAX_BITS not set to at least 2 times the keySize (Example when using 4096-bit key set FP_MAX_BITS to 8192 or greater value)

Example

```
ret = wc_InitRsaKey(&key, NULL);
if (ret == 0) {
    ret = wc_InitRng(&rng);
} else return -1;
if (ret == 0) {
    ret = wc_RsaSetRNG(&key, &rng);
} else return -1;
if (ret == 0) {
        ret = wc_MakeRsaKey(&key, 2048, WC_RSA_EXPONENT, &rng);
} else return -1;
ret = wc_RsaPSS_Sign((byte*)szMessage, (word32)XSTRLEN(szMessage)+1,
        pSignature, sizeof(pSignature),
        WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key, &rng);
if (ret > 0 ){
    sz = ret;
} else return -1;

ret = wc_RsaPSS_Verify(pSignature, sz, pt, outLen,
        WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key);
if (ret <= 0)return -1;

wc_FreeRsaKey(&key);
wc_FreeRng(&rng);
```

C.49.2.14 function wc_RsaPSS_VerifyInline

```
int wc_RsaPSS_VerifyInline(
    byte * in,
    word32 inLen,
    byte ** out,
    enum wc_HashType hash,
    int mgf,
    RsaKey * key
)
```

Decrypt input signature to verify that the message was signed by RSA key. The output uses the same byte array as the input. The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.

Parameters:

- **in** Byte array to be decrypted.

- **inLen** Length of the buffer input.
- **out** Pointer to address containing the PSS data.
- **hash** The hash type to be in message
- **mgf** Mask Generation Function Identifiers
- **key** RsaKey to use.

See:

- wc_RsaPSS_Verify
- wc_RsaPSS_Sign
- wc_RsaPSS_VerifyCheck
- wc_RsaPSS_VerifyCheck_ex
- wc_RsaPSS_VerifyCheckInline
- wc_RsaPSS_VerifyCheckInline_ex
- wc_RsaPSS_CheckPadding
- wc_RsaPSS_CheckPadding_ex
- wc_RsaSetRNG

Return:

- 0 Length of text.
- <0 An error occurred.

Example

```
ret = wc_InitRsaKey(&key, NULL);
if (ret == 0) {
    ret = wc_InitRng(&rng);
} else return -1;
if (ret == 0) {
    ret = wc_RsaSetRNG(&key, &rng);
} else return -1;
if (ret == 0) {
        ret = wc_MakeRsaKey(&key, 2048, WC_RSA_EXPONENT, &rng);
} else return -1;
ret = wc_RsaPSS_Sign(digest, digestSz, pSignature, pSignatureSz,
        WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key, &rng);
if (ret > 0 ){
    sz = ret;
} else return -1;

ret = wc_RsaPSS_VerifyInline(pSignature, sz, pt,
        WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key);
if (ret <= 0)return -1;

wc_FreeRsaKey(&key);
wc_FreeRng(&rng);
```

C.49.2.15 function wc_RsaPSS_VerifyCheck

```
int wc_RsaPSS_VerifyCheck(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    const byte * digest,
    word32 digestLen,
```

```
    enum wc_HashType hash,
    int mgf,
    RsaKey * key
)
```

Verify the message signed with RSA-PSS. Salt length is equal to hash length. The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.

Parameters:

- **in** The byte array to be decrypted.
- **inLen** The length of in.
- **out** Pointer to address containing the PSS data. Zeroed when a crypto callback device reported only a verdict (see
- **outLen** The length of out.
- **digest** Hash of the data that is being verified.
- **digestLen** Length of hash.
- **hash** Hash algorithm.
- **mgf** Mask generation function.
- **key** Public RSA key.

See:

- wc_RsaPSS_Sign
- wc_RsaPSS_Verify
- wc_RsaPSS_VerifyCheck_ex
- wc_RsaPSS_VerifyCheckInline
- wc_RsaPSS_VerifyCheckInline_ex
- wc_RsaPSS_CheckPadding
- wc_RsaPSS_CheckPadding_ex
- wc_RsaSetRNG

Return:

- the length of the PSS data on success and negative indicates failure. A crypto callback device that returns recovered data fills out and the return is that length. A device that reports only a verdict recovers nothing: out is zeroed and the return is the length the data would have been, so callers must not read out on that path.
- MEMORY_E memory exception.
-).

Example

```
ret = wc_InitRsaKey(&key, NULL);
if (ret == 0) {
    ret = wc_InitRng(&rng);
} else return -1;
if (ret == 0) {
    ret = wc_RsaSetRNG(&key, &rng);
} else return -1;
if (ret == 0) {
        ret = wc_MakeRsaKey(&key, 2048, WC_RSA_EXPONENT, &rng);
} else return -1;

if (ret == 0) {
    digestSz = wc_HashGetDigestSize(WC_HASH_TYPE_SHA256);
    ret = wc_Hash(WC_HASH_TYPE_SHA256, message, sz, digest, digestSz);
} else return -1;
```

```
if (ret == 0) {
    ret = wc_RsaPSS_Sign(digest, digestSz, pSignature, pSignatureSz,
            WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key, &rng);
    if (ret > 0 ){
        sz = ret;
    } else return -1;
} else return -1;
if (ret == 0) {
    ret = wc_RsaPSS_VerifyCheck(pSignature, sz, pt, outLen,
            digest, digestSz, WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key);
        if (ret <= 0) return -1;
} else return -1;

wc_FreeRsaKey(&key);
wc_FreeRng(&rng);
```

C.49.2.16 function wc_RsaPSS_VerifyCheck_ex

```
int wc_RsaPSS_VerifyCheck_ex(
    byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    const byte * digest,
    word32 digestLen,
    enum wc_HashType hash,
    int mgf,
    int saltLen,
    RsaKey * key
)
```

Verify the message signed with RSA-PSS. The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.

Parameters:

- **in** The byte array to be decrypted.
- **inLen** The length of in.
- **out** Pointer to address containing the PSS data.
- **outLen** The length of out.
- **digest** Hash of the data that is being verified.
- **digestLen** Length of hash.
- **hash** Hash algorithm.
- **mgf** Mask generation function.
- **saltLen** Length of salt used. RSA_PSS_SALT_LEN_DEFAULT (-1) indicates salt length is the same as the hash length. RSA_PSS_SALT_LEN_DISCOVER indicates salt length is determined from the data.
- **key** Public RSA key.

See:

- wc_RsaPSS_Sign
- wc_RsaPSS_Verify
- wc_RsaPSS_VerifyCheck
- wc_RsaPSS_VerifyCheckInline
- wc_RsaPSS_VerifyCheckInline_ex
- wc_RsaPSS_CheckPadding

- wc_RsaPSS_CheckPadding_ex
- wc_RsaSetRNG

Return:

- the length of the PSS data on success and negative indicates failure.
- MEMORY_E memory exception.

Example

```
ret = wc_InitRsaKey(&key, NULL);
if (ret == 0) {
    ret = wc_InitRng(&rng);
} else return -1;
if (ret == 0) {
    ret = wc_RsaSetRNG(&key, &rng);
} else return -1;
if (ret == 0) {
        ret = wc_MakeRsaKey(&key, 2048, WC_RSA_EXPONENT, &rng);
} else return -1;

if (ret == 0) {
    digestSz = wc_HashGetDigestSize(WC_HASH_TYPE_SHA256);
    ret = wc_Hash(WC_HASH_TYPE_SHA256, message, sz, digest, digestSz);
} else return -1;

if (ret == 0) {
    ret = wc_RsaPSS_Sign(digest, digestSz, pSignature, pSignatureSz,
            WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key, &rng);
    if (ret > 0 ){
        sz = ret;
    } else return -1;
} else return -1;
if (ret == 0) {
    ret = wc_RsaPSS_VerifyCheck_ex(pSignature, sz, pt, outLen,
            digest, digestSz, WC_HASH_TYPE_SHA256, WC_MGF1SHA256, saltLen,
 ↪  &key);
        if (ret <= 0) return -1;
} else return -1;

wc_FreeRsaKey(&key);
wc_FreeRng(&rng);
```

C.49.2.17 function wc_RsaPSS_VerifyCheckInline

```
int wc_RsaPSS_VerifyCheckInline(
    byte * in,
    word32 inLen,
    byte ** out,
    const byte * digest,
    word32 digentLen,
    enum wc_HashType hash,
    int mgf,
    RsaKey * key
)
```

Verify the message signed with RSA-PSS. The input buffer is reused for the output buffer. Salt length is equal to hash length.

Parameters:

- **in** The byte array to be decrypted.
- **inLen** The length of in.
- **out** The byte array for the decrypted data to be stored. Set to NULL when a crypto callback device reported only a verdict (see
- **digest** Hash of the data that is being verified.
- **digestLen** Length of hash.
- **hash** The hash type to be in message
- **mgf** Mask Generation Function Identifiers
- **key** The key to use for verification.

See:

- wc_RsaPSS_Sign
- wc_RsaPSS_Verify
- wc_RsaPSS_VerifyCheck
- wc_RsaPSS_VerifyCheck_ex
- wc_RsaPSS_VerifyCheckInline_ex
- wc_RsaPSS_CheckPadding
- wc_RsaPSS_CheckPadding_ex
- wc_RsaSetRNG

Return:

- the length of the PSS data on success and negative indicates failure. A crypto callback device that returns recovered data points *out into in and the return is that length. A device that reports only a verdict recovers nothing:* out is set to NULL though the return stays positive, so callers must check *out before dereferencing it.
-).

The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.

Example

```
ret = wc_InitRsaKey(&key, NULL);
if (ret == 0) {
    ret = wc_InitRng(&rng);
} else return -1;
if (ret == 0) {
    ret = wc_RsaSetRNG(&key, &rng);
} else return -1;
if (ret == 0) {
        ret = wc_MakeRsaKey(&key, 2048, WC_RSA_EXPONENT, &rng);
} else return -1;

if (ret == 0) {
    digestSz = wc_HashGetDigestSize(WC_HASH_TYPE_SHA256);
    ret = wc_Hash(WC_HASH_TYPE_SHA256, message, sz, digest, digestSz);
} else return -1;

if (ret == 0) {
    ret = wc_RsaPSS_Sign(digest, digestSz, pSignature, pSignatureSz,
            WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key, &rng);
    if (ret > 0 ){
        sz = ret;
```

```
    } else return -1;
} else return -1;
if (ret == 0) {
    ret = wc_RsaPSS_VerifyCheckInline(pSignature, sz, pt,
            digest, digestSz, WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key);
        if (ret <= 0) return -1;
} else return -1;

wc_FreeRsaKey(&key);
wc_FreeRng(&rng);
```

C.49.2.18 function wc_RsaPSS_VerifyCheckInline_ex

```
int wc_RsaPSS_VerifyCheckInline_ex(
    byte * in,
    word32 inLen,
    byte ** out,
    const byte * digest,
    word32 digentLen,
    enum wc_HashType hash,
    int mgf,
    int saltLen,
    RsaKey * key
)
```

Verify the message signed with RSA-PSS. The input buffer is reused for the output buffer. The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.

Parameters:

- **in** The byte array to be decrypted.
- **inLen** The length of in.
- **out** The byte array for the decrypted data to be stored.
- **digest** Hash of the data that is being verified.
- **digestLen** Length of hash.
- **hash** The hash type to be in message
- **mgf** Mask Generation Function Identifiers
- **saltLen** Length of salt used. RSA_PSS_SALT_LEN_DEFAULT (-1) indicates salt length is the same as the hash length. RSA_PSS_SALT_LEN_DISCOVER indicates salt length is determined from the data.
- **key** The key to use for verification.

See:

- wc_RsaPSS_Sign
- wc_RsaPSS_Verify
- wc_RsaPSS_VerifyCheck
- wc_RsaPSS_VerifyCheck_ex
- wc_RsaPSS_VerifyCheckInline
- wc_RsaPSS_CheckPadding
- wc_RsaPSS_CheckPadding_ex
- wc_RsaSetRNG

Return: the length of the PSS data on success and negative indicates failure.

Example

```
ret = wc_InitRsaKey(&key, NULL);
if (ret == 0) {
    ret = wc_InitRng(&rng);
} else return -1;
if (ret == 0) {
    ret = wc_RsaSetRNG(&key, &rng);
} else return -1;
if (ret == 0) {
        ret = wc_MakeRsaKey(&key, 2048, WC_RSA_EXPONENT, &rng);
} else return -1;

if (ret == 0) {
    digestSz = wc_HashGetDigestSize(WC_HASH_TYPE_SHA256);
    ret = wc_Hash(WC_HASH_TYPE_SHA256, message, sz, digest, digestSz);
} else return -1;

if (ret == 0) {
    ret = wc_RsaPSS_Sign(digest, digestSz, pSignature, pSignatureSz,
            WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key, &rng);
    if (ret > 0 ){
        sz = ret;
    } else return -1;
} else return -1;
if (ret == 0) {
    ret = wc_RsaPSS_VerifyCheckInline_ex(pSignature, sz, pt,
            digest, digestSz, WC_HASH_TYPE_SHA256, WC_MGF1SHA256, saltLen,
 ↪  &key);
        if (ret <= 0) return -1;
} else return -1;

wc_FreeRsaKey(&key);
wc_FreeRng(&rng);
```

C.49.2.19 function wc_RsaPSS_CheckPadding

```
int wc_RsaPSS_CheckPadding(
    const byte * in,
    word32 inLen,
    const byte * sig,
    word32 sigSz,
    enum wc_HashType hashType
)
```

Checks the PSS data to ensure that the signature matches. Salt length is equal to hash length. The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.

Parameters:

- **in** Hash of the data that is being verified.
- **inSz** Length of hash.
- **sig** Buffer holding PSS data.
- **sigSz** Size of PSS data.
- **hashType** Hash algorithm.

See:

- wc_RsaPSS_Sign

- wc_RsaPSS_Verify
- wc_RsaPSS_VerifyInline
- wc_RsaPSS_VerifyCheck
- wc_RsaPSS_VerifyCheck_ex
- wc_RsaPSS_VerifyCheckInline
- wc_RsaPSS_VerifyCheckInline_ex
- wc_RsaPSS_CheckPadding_ex
- wc_RsaSetRNG

Return:

- BAD_PADDING_E when the PSS data is invalid, BAD_FUNC_ARG when NULL is passed in to in or sig or inSz is not the same as the hash algorithm length and 0 on success.
- MEMORY_E memory exception.

Example

```
ret = wc_InitRsaKey(&key, NULL);
if (ret == 0) {
    ret = wc_InitRng(&rng);
} else return -1;
if (ret == 0) {
    ret = wc_RsaSetRNG(&key, &rng);
} else return -1;
if (ret == 0) {
        ret = wc_MakeRsaKey(&key, 2048, WC_RSA_EXPONENT, &rng);
} else return -1;
if (ret == 0) {
    digestSz = wc_HashGetDigestSize(WC_HASH_TYPE_SHA256);
    ret = wc_Hash(WC_HASH_TYPE_SHA256, message, sz, digest, digestSz);
} else return -1;
ret = wc_RsaPSS_Sign(digest, digestSz, pSignature, sizeof(pSignature),
        WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key, &rng);
if (ret > 0 ){
    sz = ret;
} else return -1;

verify = wc_RsaPSS_Verify(pSignature, sz, out, outLen,
        WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key);
if (verify <= 0)return -1;

ret = wc_RsaPSS_CheckPadding(digest, digestSz, out, verify, hash);

wc_FreeRsaKey(&key);
wc_FreeRng(&rng);
```

C.49.2.20 function wc_RsaPSS_CheckPadding_ex

```
int wc_RsaPSS_CheckPadding_ex(
    const byte * in,
    word32 inLen,
    const byte * sig,
    word32 sigSz,
    enum wc_HashType hashType,
    int saltLen,
    int bits
```

)

Checks the PSS data to ensure that the signature matches. Salt length is equal to hash length.

Parameters:

- **in** Hash of the data that is being verified.
- **inSz** Length of hash.
- **sig** Buffer holding PSS data.
- **sigSz** Size of PSS data.
- **hashType** Hash algorithm.
- **saltLen** Length of salt used. RSA_PSS_SALT_LEN_DEFAULT (-1) indicates salt length is the same as the hash length. RSA_PSS_SALT_LEN_DISCOVER indicates salt length is determined from the data.
- **bits** Can be used to calculate salt size in FIPS case

See:

- wc_RsaPSS_Sign
- wc_RsaPSS_Verify
- wc_RsaPSS_VerifyInline
- wc_RsaPSS_VerifyCheck
- wc_RsaPSS_VerifyCheck_ex
- wc_RsaPSS_VerifyCheckInline
- wc_RsaPSS_VerifyCheckInline_ex
- wc_RsaPSS_CheckPadding

Return:

- BAD_PADDING_E when the PSS data is invalid, BAD_FUNC_ARG when NULL is passed in to in or sig or inSz is not the same as the hash algorithm length and 0 on success.
- MEMORY_E memory exception.

Example

```
ret = wc_InitRsaKey(&key, NULL);
if (ret == 0) {
    ret = wc_InitRng(&rng);
} else return -1;
if (ret == 0) {
    ret = wc_RsaSetRNG(&key, &rng);
} else return -1;
if (ret == 0) {
        ret = wc_MakeRsaKey(&key, 2048, WC_RSA_EXPONENT, &rng);
} else return -1;
if (ret == 0) {
    digestSz = wc_HashGetDigestSize(WC_HASH_TYPE_SHA256);
    ret = wc_Hash(WC_HASH_TYPE_SHA256, message, sz, digest, digestSz);
} else return -1;
ret = wc_RsaPSS_Sign(digest, digestSz, pSignature, sizeof(pSignature),
        WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key, &rng);
if (ret > 0 ){
    sz = ret;
} else return -1;

verify = wc_RsaPSS_Verify(pSignature, sz, out, outLen,
        WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key);
if (verify <= 0)return -1;
```

```
ret = wc_RsaPSS_CheckPadding_ex(digest, digestSz, out, verify, hash, saltLen,
 ↪  0);

wc_FreeRsaKey(&key);
wc_FreeRng(&rng);
```

C.49.2.21 function wc_RsaEncryptSize

```
int wc_RsaEncryptSize(
    const RsaKey * key
)
```

Returns the encryption size for the provided key structure.

Parameters:

- **key** The key to use for verification.

See:

- wc_InitRsaKey
- wc_InitRsaKey_ex
- wc_MakeRsaKey

Return: Success Encryption size for the provided key structure.

Example

```
int sz = wc_RsaEncryptSize(&key);
```

C.49.2.22 function wc_RsaPrivateKeyDecode

```
int wc_RsaPrivateKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    RsaKey * key,
    word32 inSz
)
```

This function parses a DER-formatted RSA private key, extracts the private key and stores it in the given RsaKey structure. It also sets the distance parsed in idx.

Parameters:

- **input** pointer to the buffer containing the DER formatted private key to decode
- **inOutIdx** pointer to the index in the buffer at which the key begins (usually 0). As a side effect of this function, inOutIdx will store the distance parsed through the input buffer
- **key** pointer to the RsaKey structure in which to store the decoded private key
- **inSz** size of the input buffer

See:

- wc_RsaPublicKeyDecode
- wc_MakeRsaKey

Return:

- 0 Returned upon successfully parsing the private key from the DER encoded input
- ASN_PARSE_E Returned if there is an error parsing the private key from the input buffer. This may happen if the input private key is not properly formatted according to ASN.1 standards
- ASN_RSA_KEY_E Returned if there is an error reading the private key elements of the RSA key input

Example

```
RsaKey enc;
word32 idx = 0;
int ret = 0;
byte der[] = { // initialize with DER-encoded RSA private key };

wc_InitRsaKey(&enc, NULL); // not using heap hint. No custom memory
ret = wc_RsaPrivateKeyDecode(der, &idx, &enc, sizeof(der));
if( ret != 0 ) {
    // error parsing private key
}
```

C.49.2.23 function wc_RsaPublicKeyDecode

```
int wc_RsaPublicKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    RsaKey * key,
    word32 inSz
)
```

This function parses a DER-formatted RSA public key, extracts the public key and stores it in the given RsaKey structure. It also sets the distance parsed in idx.

Parameters:

- **input** pointer to the buffer containing the input DER-encoded RSA public key to decode
- **inOutIdx** pointer to the index in the buffer at which the key begins (usually 0). As a side effect of this function, inOutIdx will store the distance parsed through the input buffer
- **key** pointer to the RsaKey structure in which to store the decoded public key
- **inSz** size of the input buffer

See: wc_RsaPublicKeyDecodeRaw

Return:

- 0 Returned upon successfully parsing the public key from the DER encoded input
- ASN_PARSE_E Returned if there is an error parsing the public key from the input buffer. This may happen if the input public key is not properly formatted according to ASN.1 standards
- ASN_OBJECT_ID_E Returned if the ASN.1 Object ID does not match that of a RSA public key
- ASN_EXPECT_0_E Returned if the input key is not correctly formatted according to ASN.1 standards
- ASN_BITSTR_E Returned if the input key is not correctly formatted according to ASN.1 standards
- ASN_RSA_KEY_E Returned if there is an error reading the public key elements of the RSA key input

Example

```
RsaKey pub;
word32 idx = 0;
int ret = 0;
byte der[] = { // initialize with DER-encoded RSA public key };

wc_InitRsaKey(&pub, NULL); // not using heap hint. No custom memory
ret = wc_RsaPublicKeyDecode(der, &idx, &pub, sizeof(der));
if( ret != 0 ) {
    // error parsing public key
}
```

C.49.2.24 function wc_RsaPublicKeyDecodeRaw

```
int wc_RsaPublicKeyDecodeRaw(
    const byte * n,
    word32 nSz,
    const byte * e,
    word32 eSz,
    RsaKey * key
)
```

This function decodes the raw elements of an RSA public key, taking in the public modulus (n) and exponent (e). It stores these raw elements in the provided RsaKey structure, allowing one to use them in the encryption/decryption process.

Parameters:

- **n** pointer to a buffer containing the raw modulus parameter of the public RSA key
- **nSz** size of the buffer containing n
- **e** pointer to a buffer containing the raw exponent parameter of the public RSA key
- **eSz** size of the buffer containing e
- **key** pointer to the RsaKey struct to initialize with the provided public key elements

See: wc_RsaPublicKeyDecode

Return:

- 0 Returned upon successfully decoding the raw elements of the public key into the RsaKey structure
- BAD_FUNC_ARG Returned if any of the input arguments evaluates to NULL
- MP_INIT_E Returned if there is an error initializing an integer for use with the multiple precision integer (mp_int) library
- ASN_GETINT_E Returned if there is an error reading one of the provided RSA key elements, n or e

Example

```
RsaKey pub;
int ret = 0;
byte n[] = { // initialize with received n component of public key };
byte e[] = { // initialize with received e component of public key };

wc_InitRsaKey(&pub, NULL); // not using heap hint. No custom memory
ret = wc_RsaPublicKeyDecodeRaw(n, sizeof(n), e, sizeof(e), &pub);
if( ret != 0 ) {
    // error parsing public key elements
}
```

C.49.2.25 function wc_RsaKeyToDer

```
int wc_RsaKeyToDer(
    RsaKey * key,
    byte * output,
    word32 outLen
)
```

This function converts an RsaKey key to DER format. The result is written to output and it returns the number of bytes written.

Parameters:

- **key** Initialized RsaKey structure.
- **output** Pointer to output buffer.
- **outLen** Size of output buffer.

See:

- wc_RsaKeyToPublicDer
- wc_InitRsaKey
- wc_MakeRsaKey
- wc_InitRng

Return:

- 0 Success, number of bytes written.
- BAD_FUNC_ARG Returned if key or output is null, or if key->type is not RSA_PRIVATE, or if outLen isn't large enough for output buffer.
- MEMORY_E Returned if there is an error allocating memory.

Example

```
byte* der;
// Allocate memory for der
int derSz = // Amount of memory allocated for der;
RsaKey key;
WC_RNG rng;
long e = 65537; // standard value to use for exponent
ret = wc_MakeRsaKey(&key, 2048, e, &rng); // generate 2048 bit long
private key
wc_InitRsaKey(&key, NULL);
wc_InitRng(&rng);
if(wc_RsaKeyToDer(&key, der, derSz) != 0)
{
    // Handle the error thrown
}
```

C.49.2.26 function wc_RsaPublicEncrypt_ex

```
int wc_RsaPublicEncrypt_ex(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    RsaKey * key,
    WC_RNG * rng,
    int type,
    enum wc_HashType hash,
    int mgf,
    byte * label,
    word32 labelSz
)
```

This function performs RSA encrypt while allowing the choice of which padding to use.

Parameters:

- **in** pointer to the buffer for encryption
- **inLen** length of the buffer to encrypt
- **out** encrypted msg created

- **outLen** length of buffer available to hold encrypted msg
- **key** initialized RSA key struct
- **rng** initialized WC_RNG struct
- **type** type of padding to use (WC_RSA_OAEP_PAD or WC_RSA_PKCSV15_PAD)
- **hash** type of hash to use (choices can be found in hash.h)
- **mgf** type of mask generation function to use
- **label** an optional label to associate with encrypted message
- **labelSz** size of the optional label used

See:

- wc_RsaPublicEncrypt
- wc_RsaPrivateDecrypt_ex

Return:

- size On successfully encryption the size of the encrypted buffer is returned
- RSA_BUFFER_E RSA buffer error, output too small or input too large

Example

```
WC_RNG rng;
RsaKey key;
byte in[] = “I use Turing Machines to ask questions”
byte out[256];
int ret;
…

ret = wc_RsaPublicEncrypt_ex(in, sizeof(in), out, sizeof(out), &key, &rng,
WC_RSA_OAEP_PAD, WC_HASH_TYPE_SHA, WC_MGF1SHA1, NULL, 0);
if (ret < 0) {
    //handle error
}
```

C.49.2.27 function wc_RsaPrivateDecrypt_ex

```
int wc_RsaPrivateDecrypt_ex(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    RsaKey * key,
    int type,
    enum wc_HashType hash,
    int mgf,
    byte * label,
    word32 labelSz
)
```

This function uses RSA to decrypt a message and gives the option of what padding type.

Parameters:

- **in** pointer to the buffer for decryption
- **inLen** length of the buffer to decrypt
- **out** decrypted msg created
- **outLen** length of buffer available to hold decrypted msg
- **key** initialized RSA key struct
- **type** type of padding to use (WC_RSA_OAEP_PAD or WC_RSA_PKCSV15_PAD)

- **hash** type of hash to use (choices can be found in hash.h)
- **mgf** type of mask generation function to use
- **label** an optional label to associate with encrypted message
- **labelSz** size of the optional label used

See: none

Return:

- size On successful decryption, the size of the decrypted message is returned.
- MEMORY_E Returned if not enough memory on system to malloc a needed array.
- BAD_FUNC_ARG Returned if a bad argument was passed into the function.

Example

```
WC_RNG rng;
RsaKey key;
byte in[] = “I use Turing Machines to ask questions”
byte out[256];
byte plain[256];
int ret;
…
ret = wc_RsaPublicEncrypt_ex(in, sizeof(in), out, sizeof(out), &key,
&rng, WC_RSA_OAEP_PAD, WC_HASH_TYPE_SHA, WC_MGF1SHA1, NULL, 0);
if (ret < 0) {
    //handle error
}
…
ret = wc_RsaPrivateDecrypt_ex(out, ret, plain, sizeof(plain), &key,
WC_RSA_OAEP_PAD, WC_HASH_TYPE_SHA, WC_MGF1SHA1, NULL, 0);

if (ret < 0) {
    //handle error
}
```

C.49.2.28 function wc_RsaPrivateDecryptInline_ex

```
int wc_RsaPrivateDecryptInline_ex(
    byte * in,
    word32 inLen,
    byte ** out,
    RsaKey * key,
    int type,
    enum wc_HashType hash,
    int mgf,
    byte * label,
    word32 labelSz
)
```

This function uses RSA to decrypt a message inline and gives the option of what padding type. The in buffer will contain the decrypted message after being called and the out byte pointer will point to the location in the “in” buffer where the plain text is.

Parameters:

- **in** pointer to the buffer for decryption
- **inLen** length of the buffer to decrypt
- **out** pointer to location of decrypted message in “in” buffer

- **key** initialized RSA key struct
- **type** type of padding to use (WC_RSA_OAEP_PAD or WC_RSA_PKCSV15_PAD)
- **hash** type of hash to use (choices can be found in hash.h)
- **mgf** type of mask generation function to use
- **label** an optional label to associate with encrypted message
- **labelSz** size of the optional label used

See: none

Return:

- size On successful decryption, the size of the decrypted message is returned.
- MEMORY_E: Returned if not enough memory on system to malloc a needed array.
- RSA_PAD_E: Returned if an error in the padding was encountered.
- BAD_PADDING_E: Returned if an error happened during parsing past padding.
- BAD_FUNC_ARG: Returned if a bad argument was passed into the function.

Example

```
WC_RNG rng;
RsaKey key;
byte in[] = “I use Turing Machines to ask questions”
byte out[256];
byte* plain;
int ret;
…
ret = wc_RsaPublicEncrypt_ex(in, sizeof(in), out, sizeof(out), &key,
&rng, WC_RSA_OAEP_PAD, WC_HASH_TYPE_SHA, WC_MGF1SHA1, NULL, 0);

if (ret < 0) {
    //handle error
}
…
ret = wc_RsaPrivateDecryptInline_ex(out, ret, &plain, &key,
WC_RSA_OAEP_PAD, WC_HASH_TYPE_SHA, WC_MGF1SHA1, NULL, 0);

if (ret < 0) {
    //handle error
}
```

C.49.2.29 function wc_RsaFlattenPublicKey

```
int wc_RsaFlattenPublicKey(
    const RsaKey * key,
    byte * e,
    word32 * eSz,
    byte * n,
    word32 * nSz
)
```

Flattens the RsaKey structure into individual elements (e, n) used for the RSA algorithm.

Parameters:

- **key** The key to use for verification.
- **e** a buffer for the value of e. e is a large positive integer in the RSA modular arithmetic operation.
- **eSz** the size of the e buffer.
- **n** a buffer for the value of n. n is a large positive integer in the RSA modular arithmetic operation.

- **nSz** the size of the n buffer.

See:

- wc_InitRsaKey
- wc_InitRsaKey_ex
- wc_MakeRsaKey

Return:

- 0 Returned if the function executed normally, without error.
- BAD_FUNC_ARG: Returned if any of the parameters are passed in with a null value.
- RSA_BUFFER_E: Returned if the e or n buffers passed in are not the correct size.
- MP_MEM: Returned if an internal function has memory errors.
- MP_VAL: Returned if an internal function argument is not valid.

Example

```
Rsa key; // A valid RSA key.
byte e[ buffer sz E.g. 256 ];
byte n[256];
int ret;
word32 eSz = sizeof(e);
word32 nSz = sizeof(n);
...
ret = wc_RsaFlattenPublicKey(&key, e, &eSz, n, &nSz);
if (ret != 0) {
    // Failure case.
}
```

C.49.2.30 function wc_RsaKeyToPublicDer

```
int wc_RsaKeyToPublicDer(
    RsaKey * key,
    byte * output,
    word32 outLen
)
```

Convert Rsa Public key to DER format. Writes to output, and returns count of bytes written.

Parameters:

- **key** The RSA key structure to convert.
- **output** Output buffer to hold DER. (if NULL will return length only)
- **outLen** Length of buffer.

See:

- wc_RsaPublicKeyDerSize
- wc_RsaKeyToPublicDer_ex
- wc_InitRsaKey

Return:

- 0 Success, number of bytes written.
- BAD_FUNC_ARG Returned if key or output is null.
- MEMORY_E Returned when an error allocating memory occurs.
- <0 Error

Example

```
RsaKey key;

wc_InitRsaKey(&key, NULL);
// Use key

const int BUFFER_SIZE = 1024; // Some adequate size for the buffer
byte output[BUFFER_SIZE];
if (wc_RsaKeyToPublicDer(&key, output, sizeof(output)) != 0) {
    // Handle Error
}
```

C.49.2.31 function wc_RsaKeyToPublicDer_ex

```
int wc_RsaKeyToPublicDer_ex(
    RsaKey * key,
    byte * output,
    word32 outLen,
    int with_header
)
```

Convert RSA Public key to DER format. Writes to output, and returns count of bytes written. If with_header is 0 then only the (seq + n + e) is returned in ASN.1 DER format and will exclude the header.

Parameters:

- **key** The RSA key structure to convert.
- **output** Output buffer to hold DER. (if NULL will return length only)
- **outLen** Length of buffer.

See:

- wc_RsaPublicKeyDerSize
- wc_RsaKeyToPublicDer
- wc_InitRsaKey

Return:

- 0 Success, number of bytes written.
- BAD_FUNC_ARG Returned if key or output is null.
- MEMORY_E Returned when an error allocating memory occurs.
- <0 Error

Example

```
RsaKey key;

wc_InitRsaKey(&key, NULL);
// Use key

const int BUFFER_SIZE = 1024; // Some adequate size for the buffer
byte output[BUFFER_SIZE];
if (wc_RsaKeyToPublicDer_ex(&key, output, sizeof(output), 0) != 0) {
    // Handle Error
}
```

C.49.2.32 function wc_MakeRsaKey

```
int wc_MakeRsaKey(
    RsaKey * key,
    int size,
    long e,
    WC_RNG * rng
)
```

This function generates a RSA private key of length size (in bits) and given exponent (e). It then stores this key in the provided RsaKey structure, so that it may be used for encryption/decryption. A secure number to use for e is 65537. size is required to be greater than or equal to RSA_MIN_SIZE and less than or equal to RSA_MAX_SIZE. For this function to be available, the option WOLFSSL_KEY_GEN must be enabled at compile time. This can be accomplished with –enable-keygen if using ./configure.

Parameters:

- **key** pointer to the RsaKey structure in which to store the generated private key
- **size** desired key length, in bits. Required to be greater than RSA_MIN_SIZE and less than RSA_MAX_SIZE
- **e** exponent parameter to use for generating the key. A secure choice is 65537
- **rng** pointer to an RNG structure to use for random number generation while making the key

See: none

Return:

- 0 Returned upon successfully generating a RSA private key
- BAD_FUNC_ARG Returned if any of the input arguments are NULL, the size parameter falls outside of the necessary bounds, or e is incorrectly chosen
- RNG_FAILURE_E Returned if there is an error generating a random block using the provided RNG structure
- MP_INIT_E
- MP_READ_E May be May be returned if there is an error in the math library used while generating the RSA key returned if there is an error in the math library used while generating the RSA key
- MP_CMP_E May be returned if there is an error in the math library used while generating the RSA key
- MP_INVMOD_E May be returned if there is an error in the math library used while generating the RSA key
- MP_EXPTMOD_E May be returned if there is an error in the math library used while generating the RSA key
- MP_MOD_E May be returned if there is an error in the math library used while generating the RSA key
- MP_MUL_E May be returned if there is an error in the math library used while generating the RSA key
- MP_ADD_E May be returned if there is an error in the math library used while generating the RSA key
- MP_MULMOD_E May be returned if there is an error in the math library used while generating the RSA key
- MP_TO_E May be returned if there is an error in the math library used while generating the RSA key
- MP_MEM May be returned if there is an error in the math library used while generating the RSA key
- MP_ZERO_E May be returned if there is an error in the math library used while generating the RSA key

Example

```
RsaKey priv;
WC_RNG rng;
int ret = 0;
long e = 65537; // standard value to use for exponent

wc_InitRsaKey(&priv, NULL); // not using heap hint. No custom memory
wc_InitRng(&rng);
// generate 2048 bit long private key
ret = wc_MakeRsaKey(&priv, 2048, e, &rng);
if( ret != 0 ) {
    // error generating private key
}
```

C.49.2.33 function wc_RsaSetNonBlock

```
int wc_RsaSetNonBlock(
    RsaKey * key,
    RsaNb * nb
)
```

This function sets the non-blocking RSA context. When a RsaNb context is set it enables fast math based non-blocking exptmod, which splits the RSA function into many smaller operations. Enabled when WC_RSA_NONBLOCK is defined.

Parameters:

- **key** The RSA key structure
- **nb** The RSA non-blocking structure for this RSA key to use.

See: wc_RsaSetNonBlockTime

Return:

- 0 Success
- BAD_FUNC_ARG Returned if key or nb is null.

Example

```
int ret, count = 0;
RsaKey key;
RsaNb  nb;

wc_InitRsaKey(&key, NULL);

// Enable non-blocking RSA mode - provide context
ret = wc_RsaSetNonBlock(key, &nb);
if (ret != 0)
    return ret;

do {
    ret = wc_RsaSSL_Sign(in, inLen, out, outSz, key, rng);
    count++; // track number of would blocks
    if (ret == FP_WOULDBLOCK) {
        // do "other" work here
    }
} while (ret == FP_WOULDBLOCK);
if (ret < 0) {
    return ret;
```

```
}

printf("RSA non-block sign: size %d, %d times\n", ret, count);
```

C.49.2.34 function wc_RsaSetNonBlockTime

```
int wc_RsaSetNonBlockTime(
    RsaKey * key,
    word32 maxBlockUs,
    word32 cpuMHz
)
```

This function configures the maximum amount of blocking time in microseconds. It uses a pre-computed table (see tfm.c exptModNbInst) along with the CPU speed in megahertz to determine if the next operation can be completed within the maximum blocking time provided. Enabled when WC_RSA_NONBLOCK_TIME is defined.

Parameters:

- **key** The RSA key structure.
- **maxBlockUs** Maximum time to block microseconds.
- **cpuMHz** CPU speed in megahertz.

See: wc_RsaSetNonBlock

Return:

- 0 Success
- BAD_FUNC_ARG Returned if key is null or wc_RsaSetNonBlock was not previously called and key->nb is null.

Example

```
RsaKey key;
RsaNb  nb;

wc_InitRsaKey(&key, NULL);
wc_RsaSetNonBlock(key, &nb);
wc_RsaSetNonBlockTime(&key, 4000, 160); // Block Max = 4 ms, CPU = 160MHz
```

C.49.2.35 function wc_InitRsaKey_ex

```
int wc_InitRsaKey_ex(
    RsaKey * key,
    void * heap,
    int devId
)
```

Initializes RSA key with heap and device ID.

Parameters:

- **key** RSA key structure
- **heap** Heap hint
- **devId** Device ID

See: wc_InitRsaKey

Return:

- 0 on success

- negative on error

Example

```
RsaKey key;
int ret = wc_InitRsaKey_ex(&key, NULL, INVALID_DEVID);
```

C.49.2.36 function wc_NewRsaKey

```
RsaKey * wc_NewRsaKey(
    void * heap,
    int devId,
    int * result_code
)
```

Allocates and initializes new RSA key. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.

Parameters:

- **heap** Heap hint
- **devId** Device ID
- **result_code** Result code pointer

See: wc_DeleteRsaKey

Return:

- RsaKey pointer on success
- NULL on failure

Note: This API is only available when WC_NO_CONSTRUCTORS is not defined. WC_NO_CONSTRUCTORS is automatically defined when WOLFSSL_NO_MALLOC is defined.

Example

```
int result;
RsaKey* key = wc_NewRsaKey(NULL, INVALID_DEVID, &result);
```

C.49.2.37 function wc_DeleteRsaKey

```
int wc_DeleteRsaKey(
    RsaKey * key,
    RsaKey ** key_p
)
```

Deletes and frees RSA key. These New/Delete functions are exposed to support allocation of the structure using dynamic memory to provide better ABI compatibility.

Parameters:

- **key** RSA key to delete
- **key_p** Pointer to key pointer

See: wc_NewRsaKey

Return:

- 0 on success
- negative on error

Note: This API is only available when WC_NO_CONSTRUCTORS is not defined. WC_NO_CONSTRUCTORS is automatically defined when WOLFSSL_NO_MALLOC is defined.

Example

```
RsaKey* key;
int ret = wc_DeleteRsaKey(key, &key);
```

C.49.2.38 function wc_InitRsaKey_Label

```
int wc_InitRsaKey_Label(
    RsaKey * key,
    const char * label,
    void * heap,
    int devId
)
```

Initializes RSA key with label.

Parameters:

- **key** RSA key structure
- **label** Label string
- **heap** Heap hint
- **devId** Device ID

See: wc_InitRsaKey_ex

Return:

- 0 on success
- negative on error

Note: This API is only available when WOLF_PRIVATE_KEY_ID is defined, which is set for PKCS11 support.

Example

```
RsaKey key;
int ret = wc_InitRsaKey_Label(&key, "mykey", NULL,
                              INVALID_DEVID);
```

C.49.2.39 function wc_CheckRsaKey

```
int wc_CheckRsaKey(
    RsaKey * key
)
```

Checks RSA key validity.

Parameters:

- **key** RSA key to check

See: wc_MakeRsaKey

Return:

- 0 on success
- negative on error

Example

```
RsaKey key;
int ret = wc_CheckRsaKey(&key);
```

C.49.2.40 function wc_RsaUseKeyId

```
int wc_RsaUseKeyId(
    RsaKey * key,
    word32 keyId,
    word32 flags
)
```

Uses key ID for hardware RSA.

Parameters:

- **key** RSA key
- **keyId** Key identifier
- **flags** Flags

See: wc_RsaGetKeyId

Return:

- 0 on success
- negative on error

Example

```
RsaKey key;
int ret = wc_RsaUseKeyId(&key, 1, 0);
```

C.49.2.41 function wc_RsaGetKeyId

```
int wc_RsaGetKeyId(
    RsaKey * key,
    word32 * keyId
)
```

Gets key ID from hardware RSA key.

Parameters:

- **key** RSA key
- **keyId** Key identifier pointer

See: wc_RsaUseKeyId

Return:

- 0 on success
- negative on error

Example

```
RsaKey key;
word32 keyId;
int ret = wc_RsaGetKeyId(&key, &keyId);
```

C.49.2.42 function wc_RsaFunction

```
int wc_RsaFunction(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen,
    int type,
    RsaKey * key,
    WC_RNG * rng
)
```

Performs RSA operation.

Parameters:

- **in** Input buffer
- **inLen** Input length
- **out** Output buffer
- **outLen** Output length pointer
- **type** Operation type
- **key** RSA key
- **rng** Random number generator

See: wc_RsaPublicEncrypt

Return:

- 0 on success
- negative on error

Example

```
RsaKey key;
WC_RNG rng;
byte in[256], out[256];
word32 outLen = sizeof(out);
int ret = wc_RsaFunction(in, 256, out, &outLen,
                         RSA_PUBLIC_ENCRYPT, &key, &rng);
```

C.49.2.43 function wc_RsaPSS_Sign_ex

```
int wc_RsaPSS_Sign_ex(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    enum wc_HashType hash,
    int mgf,
    int saltLen,
    RsaKey * key,
    WC_RNG * rng
)
```

Signs with RSA-PSS extended options.

Parameters:

- **in** Input buffer
- **inLen** Input length

- **out** Output buffer
- **outLen** Output buffer size
- **hash** Hash type
- **mgf** MGF type
- **saltLen** Salt length
- **key** RSA key
- **rng** Random number generator

See: wc_RsaPSS_Sign

Return:

- Size of signature on success
- negative on error

Example

```
RsaKey key;
WC_RNG rng;
byte in[32], sig[256];
int ret = wc_RsaPSS_Sign_ex(in, 32, sig, sizeof(sig),
                            WC_HASH_TYPE_SHA256,
                            WC_MGF1SHA256, 32, &key, &rng);
```

C.49.2.44 function wc_RsaSSL_Verify_ex

```
int wc_RsaSSL_Verify_ex(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    RsaKey * key,
    int pad_type
)
```

Verifies RSA signature with padding type.

Parameters:

- **in** Input signature
- **inLen** Signature length
- **out** Output buffer
- **outLen** Output buffer size
- **key** RSA key
- **pad_type** Padding type

See: wc_RsaSSL_Verify

Return:

- Size of decrypted data on success
- negative on error

Example

```
RsaKey key;
byte sig[256], out[256];
int ret = wc_RsaSSL_Verify_ex(sig, 256, out, sizeof(out),
                              &key, RSA_PKCS1_PADDING);
```

C.49.2.45 function wc_RsaSSL_Verify_ex2

```
int wc_RsaSSL_Verify_ex2(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    RsaKey * key,
    int pad_type,
    enum wc_HashType hash
)
```

Verifies RSA signature with hash type.

Parameters:

- **in** Input signature
- **inLen** Signature length
- **out** Output buffer
- **outLen** Output buffer size
- **key** RSA key
- **pad_type** Padding type
- **hash** Hash type

See: wc_RsaSSL_Verify_ex

Return:

- Size of decrypted data on success
- negative on error

Example

```
RsaKey key;
byte sig[256], out[256];
int ret = wc_RsaSSL_Verify_ex2(sig, 256, out, sizeof(out),
                               &key, RSA_PKCS1_PADDING,
                               WC_HASH_TYPE_SHA256);
```

C.49.2.46 function wc_RsaPSS_VerifyInline_ex

```
int wc_RsaPSS_VerifyInline_ex(
    byte * in,
    word32 inLen,
    byte ** out,
    enum wc_HashType hash,
    int mgf,
    int saltLen,
    RsaKey * key
)
```

Verifies RSA-PSS inline with extended options.

Parameters:

- **in** Input/output buffer
- **inLen** Input length
- **out** Output pointer
- **hash** Hash type
- **mgf** MGF type

- **saltLen** Salt length
- **key** RSA key

See: wc_RsaPSS_VerifyInline

Return:

- Size of verified data on success
- negative on error

Example

```
RsaKey key;
byte sig[256];
byte* out;
int ret = wc_RsaPSS_VerifyInline_ex(sig, 256, &out,
                                    WC_HASH_TYPE_SHA256,
                                    WC_MGF1SHA256, 32, &key);
```

C.49.2.47 function wc_RsaPSS_Verify_ex

```
int wc_RsaPSS_Verify_ex(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    enum wc_HashType hash,
    int mgf,
    int saltLen,
    RsaKey * key
)
```

Verifies RSA-PSS with extended options.

Parameters:

- **in** Input signature
- **inLen** Signature length
- **out** Output buffer
- **outLen** Output buffer size
- **hash** Hash type
- **mgf** MGF type
- **saltLen** Salt length
- **key** RSA key

See: wc_RsaPSS_Verify

Return:

- Size of verified data on success
- negative on error

Example

```
RsaKey key;
byte sig[256], out[256];
int ret = wc_RsaPSS_Verify_ex(sig, 256, out, sizeof(out),
                              WC_HASH_TYPE_SHA256,
                              WC_MGF1SHA256, 32, &key);
```

C.49.2.48 function wc_RsaPSS_CheckPadding_ex2

```
int wc_RsaPSS_CheckPadding_ex2(
    const byte * in,
    word32 inLen,
    const byte * sig,
    word32 sigSz,
    enum wc_HashType hashType,
    int saltLen,
    int bits,
    void * heap
)
```

Checks RSA-PSS padding with extended options.

Parameters:

- **in** Padded data
- **inLen** Padded data length
- **sig** Signature
- **sigSz** Signature size
- **hashType** Hash type
- **saltLen** Salt length
- **bits** Key size in bits
- **heap** Heap hint

See: wc_RsaPSS_CheckPadding_ex

Return:

- 0 on success
- negative on error

Example

```
byte padded[256], sig[256];
int ret = wc_RsaPSS_CheckPadding_ex2(padded, 256, sig, 256,
                                     WC_HASH_TYPE_SHA256, 32,
                                     2048, NULL);
```

C.49.2.49 function wc_RsaExportKey

```
int wc_RsaExportKey(
    const RsaKey * key,
    byte * e,
    word32 * eSz,
    byte * n,
    word32 * nSz,
    byte * d,
    word32 * dSz,
    byte * p,
    word32 * pSz,
    byte * q,
    word32 * qSz
)
```

Exports RSA key components.

Parameters:

- **key** RSA key
- **e** Public exponent buffer
- **eSz** Public exponent size pointer
- **n** Modulus buffer
- **nSz** Modulus size pointer
- **d** Private exponent buffer
- **dSz** Private exponent size pointer
- **p** Prime p buffer
- **pSz** Prime p size pointer
- **q** Prime q buffer
- **qSz** Prime q size pointer

See: wc_RsaFlattenPublicKey

Return:

- 0 on success
- negative on error

Example

```
RsaKey key;
byte e[3], n[256], d[256], p[128], q[128];
word32 eSz = 3, nSz = 256, dSz = 256, pSz = 128, qSz = 128;
int ret = wc_RsaExportKey(&key, e, &eSz, n, &nSz, d, &dSz,
                          p, &pSz, q, &qSz);
```

C.49.2.50 function wc_CheckProbablePrime_ex

```
int wc_CheckProbablePrime_ex(
    const byte * p,
    word32 pSz,
    const byte * q,
    word32 qSz,
    const byte * e,
    word32 eSz,
    int nlen,
    int * isPrime,
    WC_RNG * rng
)
```

Checks probable prime with extended options.

Parameters:

- **p** Prime p buffer
- **pSz** Prime p size
- **q** Prime q buffer
- **qSz** Prime q size
- **e** Public exponent buffer
- **eSz** Public exponent size
- **nlen** Modulus length
- **isPrime** Prime result pointer
- **rng** Random number generator

See: wc_CheckProbablePrime

Return:

- 0 on success

- negative on error

Example

```
byte p[128], q[128], e[3];
int isPrime;
WC_RNG rng;
int ret = wc_CheckProbablePrime_ex(p, 128, q, 128, e, 3,
                                   2048, &isPrime, &rng);
```

C.49.2.51 function wc_CheckProbablePrime

```
int wc_CheckProbablePrime(
    const byte * p,
    word32 pSz,
    const byte * q,
    word32 qSz,
    const byte * e,
    word32 eSz,
    int nlen,
    int * isPrime
)
```

Checks probable prime.

Parameters:

- **p** Prime p buffer
- **pSz** Prime p size
- **q** Prime q buffer
- **qSz** Prime q size
- **e** Public exponent buffer
- **eSz** Public exponent size
- **nlen** Modulus length
- **isPrime** Prime result pointer

See: wc_CheckProbablePrime_ex

Return:

- 0 on success
- negative on error

Example

```
byte p[128], q[128], e[3];
int isPrime;
int ret = wc_CheckProbablePrime(p, 128, q, 128, e, 3, 2048,
                                &isPrime);
```

C.49.2.52 function wc_RsaPad_ex

```
int wc_RsaPad_ex(
    const byte * input,
    word32 inputLen,
    byte * pkcsBlock,
    word32 pkcsBlockLen,
    byte padValue,
    WC_RNG * rng,
```

```
    int padType,
    enum wc_HashType hType,
    int mgf,
    byte * optLabel,
    word32 labelLen,
    int saltLen,
    int bits,
    void * heap
)
```

Pads data with extended options.

Parameters:

- **input** Input data
- **inputLen** Input length
- **pkcsBlock** Output padded block
- **pkcsBlockLen** Padded block size
- **padValue** Pad value
- **rng** Random number generator
- **padType** Padding type
- **hType** Hash type
- **mgf** MGF type
- **optLabel** Optional label
- **labelLen** Label length
- **saltLen** Salt length
- **bits** Key size in bits
- **heap** Heap hint

See: wc_RsaUnPad_ex

Return:

- 0 on success
- negative on error

Example

```
byte in[32], padded[256];
WC_RNG rng;
int ret = wc_RsaPad_ex(in, 32, padded, 256, 0x00, &rng,
                       RSA_BLOCK_TYPE_1,
                       WC_HASH_TYPE_SHA256, WC_MGF1SHA256,
                       NULL, 0, 32, 2048, NULL);
```

C.49.2.53 function wc_RsaUnPad_ex

```
int wc_RsaUnPad_ex(
    byte * pkcsBlock,
    word32 pkcsBlockLen,
    byte ** out,
    byte padValue,
    int padType,
    enum wc_HashType hType,
    int mgf,
    byte * optLabel,
    word32 labelLen,
    int saltLen,
```

```
    int bits,
    void * heap
)
```

Unpads data with extended options.

Parameters:

- **pkcsBlock** Padded block
- **pkcsBlockLen** Padded block length
- **out** Output pointer
- **padValue** Pad value
- **padType** Padding type
- **hType** Hash type
- **mgf** MGF type
- **optLabel** Optional label
- **labelLen** Label length
- **saltLen** Salt length
- **bits** Key size in bits
- **heap** Heap hint

See: wc_RsaPad_ex

Return:

- Size of unpadded data on success
- negative on error

Example

```
byte padded[256];
byte* out;
int ret = wc_RsaUnPad_ex(padded, 256, &out, 0x00,
                         RSA_BLOCK_TYPE_1,
                         WC_HASH_TYPE_SHA256, WC_MGF1SHA256,
                         NULL, 0, 32, 2048, NULL);
```

C.49.2.54 function wc_RsaPrivateKeyDecodeRaw

```
int wc_RsaPrivateKeyDecodeRaw(
    const byte * n,
    word32 nSz,
    const byte * e,
    word32 eSz,
    const byte * d,
    word32 dSz,
    const byte * u,
    word32 uSz,
    const byte * p,
    word32 pSz,
    const byte * q,
    word32 qSz,
    const byte * dP,
    word32 dPSz,
    const byte * dQ,
    word32 dQSz,
    RsaKey * key
)
```

Decodes raw RSA private key.

Parameters:

- **n** Modulus buffer
- **nSz** Modulus size
- **e** Public exponent buffer
- **eSz** Public exponent size
- **d** Private exponent buffer
- **dSz** Private exponent size
- **u** Coefficient buffer
- **uSz** Coefficient size
- **p** Prime p buffer
- **pSz** Prime p size
- **q** Prime q buffer
- **qSz** Prime q size
- **dP** dP buffer
- **dPSz** dP size
- **dQ** dQ buffer
- **dQSz** dQ size
- **key** RSA key

See: wc_RsaPrivateKeyDecode

Return:

- 0 on success
- negative on error

Example

```
RsaKey key;
byte n[256], e[3], d[256], u[256], p[128], q[128];
byte dP[128], dQ[128];
int ret = wc_RsaPrivateKeyDecodeRaw(n, 256, e, 3, d, 256,
                                    u, 256, p, 128, q, 128,
                                    dP, 128, dQ, 128, &key);
```

C.49.3 Source code

```
int  wc_InitRsaKey(RsaKey* key, void* heap);

int  wc_InitRsaKey_Id(RsaKey* key, unsigned char* id, int len,
        void* heap, int devId);

int wc_RsaSetRNG(RsaKey* key, WC_RNG* rng);

int  wc_FreeRsaKey(RsaKey* key);

int wc_RsaDirect(const byte* in, word32 inLen, byte* out, word32* outSz,
        RsaKey* key, int type, WC_RNG* rng);

int  wc_RsaPublicEncrypt(const byte* in, word32 inLen, byte* out,
                                 word32 outLen, RsaKey* key, WC_RNG* rng);

int  wc_RsaPrivateDecryptInline(byte* in, word32 inLen, byte** out,
```

```
                                        RsaKey* key);

int  wc_RsaPrivateDecrypt(const byte* in, word32 inLen, byte* out,
                                  word32 outLen, RsaKey* key);

int  wc_RsaSSL_Sign(const byte* in, word32 inLen, byte* out,
                            word32 outLen, RsaKey* key, WC_RNG* rng);

int  wc_RsaSSL_VerifyInline(byte* in, word32 inLen, byte** out,
                                    RsaKey* key);

int  wc_RsaSSL_Verify(const byte* in, word32 inLen, byte* out,
                              word32 outLen, RsaKey* key);

int  wc_RsaPSS_Sign(const byte* in, word32 inLen, byte* out,
                                word32 outLen, enum wc_HashType hash, int mgf,
                                RsaKey* key, WC_RNG* rng);

int  wc_RsaPSS_Verify(const byte* in, word32 inLen, byte* out,
                                 word32 outLen, enum wc_HashType hash, int mgf,
                                  RsaKey* key);

int  wc_RsaPSS_VerifyInline(byte* in, word32 inLen, byte** out,
                                       enum wc_HashType hash, int mgf,
                                       RsaKey* key);
int  wc_RsaPSS_VerifyCheck(const byte* in, word32 inLen,
                               byte* out, word32 outLen,
                               const byte* digest, word32 digestLen,
                               enum wc_HashType hash, int mgf,
                               RsaKey* key);
int  wc_RsaPSS_VerifyCheck_ex(byte* in, word32 inLen,
                               byte* out, word32 outLen,
                               const byte* digest, word32 digestLen,
                               enum wc_HashType hash, int mgf, int saltLen,
                               RsaKey* key);

int  wc_RsaPSS_VerifyCheckInline(byte* in, word32 inLen, byte** out,
                               const byte* digest, word32 digentLen,
                               enum wc_HashType hash, int mgf,
                               RsaKey* key);
int  wc_RsaPSS_VerifyCheckInline_ex(byte* in, word32 inLen, byte** out,
                               const byte* digest, word32 digentLen,
                               enum wc_HashType hash, int mgf, int saltLen,
                               RsaKey* key);

int  wc_RsaPSS_CheckPadding(const byte* in, word32 inLen, const byte* sig,
                                       word32 sigSz,
                                       enum wc_HashType hashType);
int  wc_RsaPSS_CheckPadding_ex(const byte* in, word32 inLen, const byte* sig,
                word32 sigSz, enum wc_HashType hashType, int saltLen, int bits);
int  wc_RsaEncryptSize(const RsaKey* key);

int  wc_RsaPrivateKeyDecode(const byte* input, word32* inOutIdx,
                            RsaKey* key, word32 inSz);
```

```
int  wc_RsaPublicKeyDecode(const byte* input, word32* inOutIdx,
                           RsaKey* key, word32 inSz);

int  wc_RsaPublicKeyDecodeRaw(const byte* n, word32 nSz,
                                     const byte* e, word32 eSz, RsaKey* key);

int wc_RsaKeyToDer(RsaKey* key, byte* output, word32 outLen);

int  wc_RsaPublicEncrypt_ex(const byte* in, word32 inLen, byte* out,
                  word32 outLen, RsaKey* key, WC_RNG* rng, int type,
                 enum wc_HashType hash, int mgf, byte* label, word32 labelSz);

int  wc_RsaPrivateDecrypt_ex(const byte* in, word32 inLen,
                  byte* out, word32 outLen, RsaKey* key, int type,
                 enum wc_HashType hash, int mgf, byte* label, word32 labelSz);

int  wc_RsaPrivateDecryptInline_ex(byte* in, word32 inLen,
                    byte** out, RsaKey* key, int type, enum wc_HashType hash,
                    int mgf, byte* label, word32 labelSz);

int  wc_RsaFlattenPublicKey(const RsaKey* key, byte* e, word32* eSz, byte* n,
                           word32* nSz);

int wc_RsaKeyToPublicDer(RsaKey* key, byte* output, word32 outLen);

int wc_RsaKeyToPublicDer_ex(RsaKey* key, byte* output, word32 outLen,
    int with_header);

int wc_MakeRsaKey(RsaKey* key, int size, long e, WC_RNG* rng);

int wc_RsaSetNonBlock(RsaKey* key, RsaNb* nb);

int wc_RsaSetNonBlockTime(RsaKey* key, word32 maxBlockUs,
    word32 cpuMHz);
int wc_InitRsaKey_ex(RsaKey* key, void* heap, int devId);

RsaKey* wc_NewRsaKey(void* heap, int devId, int *result_code);

int wc_DeleteRsaKey(RsaKey* key, RsaKey** key_p);

int wc_InitRsaKey_Label(RsaKey* key, const char* label, void* heap,
    int devId);

int wc_CheckRsaKey(RsaKey* key);

int wc_RsaUseKeyId(RsaKey* key, word32 keyId, word32 flags);

int wc_RsaGetKeyId(RsaKey* key, word32* keyId);

int wc_RsaFunction(const byte* in, word32 inLen, byte* out,
    word32* outLen, int type, RsaKey* key, WC_RNG* rng);

int wc_RsaPSS_Sign_ex(const byte* in, word32 inLen, byte* out,
```

```
    word32 outLen, enum wc_HashType hash, int mgf, int saltLen,
    RsaKey* key, WC_RNG* rng);

int wc_RsaSSL_Verify_ex(const byte* in, word32 inLen, byte* out,
    word32 outLen, RsaKey* key, int pad_type);

int wc_RsaSSL_Verify_ex2(const byte* in, word32 inLen, byte* out,
    word32 outLen, RsaKey* key, int pad_type,
    enum wc_HashType hash);

int wc_RsaPSS_VerifyInline_ex(byte* in, word32 inLen, byte** out,
    enum wc_HashType hash, int mgf, int saltLen, RsaKey* key);

int wc_RsaPSS_Verify_ex(const byte* in, word32 inLen, byte* out,
    word32 outLen, enum wc_HashType hash, int mgf, int saltLen,
    RsaKey* key);

int wc_RsaPSS_CheckPadding_ex2(const byte* in, word32 inLen,
    const byte* sig, word32 sigSz, enum wc_HashType hashType,
    int saltLen, int bits, void* heap);

int wc_RsaExportKey(const RsaKey* key, byte* e, word32* eSz,
    byte* n, word32* nSz, byte* d, word32* dSz, byte* p,
    word32* pSz, byte* q, word32* qSz);

int wc_CheckProbablePrime_ex(const byte* p, word32 pSz,
    const byte* q, word32 qSz, const byte* e, word32 eSz,
    int nlen, int* isPrime, WC_RNG* rng);

int wc_CheckProbablePrime(const byte* p, word32 pSz,
    const byte* q, word32 qSz, const byte* e, word32 eSz,
    int nlen, int* isPrime);

int wc_RsaPad_ex(const byte* input, word32 inputLen,
    byte* pkcsBlock, word32 pkcsBlockLen, byte padValue,
    WC_RNG* rng, int padType, enum wc_HashType hType, int mgf,
    byte* optLabel, word32 labelLen, int saltLen, int bits,
    void* heap);

int wc_RsaUnPad_ex(byte* pkcsBlock, word32 pkcsBlockLen,
    byte** out, byte padValue, int padType,
    enum wc_HashType hType, int mgf, byte* optLabel,
    word32 labelLen, int saltLen, int bits, void* heap);

int wc_RsaPrivateKeyDecodeRaw(const byte* n, word32 nSz,
    const byte* e, word32 eSz, const byte* d, word32 dSz,
    const byte* u, word32 uSz, const byte* p, word32 pSz,
    const byte* q, word32 qSz, const byte* dP, word32 dPSz,
    const byte* dQ, word32 dQSz, RsaKey* key);
```

C.50 sakke.h

C.50.1 Functions

	Name
int	**wc_InitSakkeKey**(SakkeKey * key, void * heap, int devId)
int	**wc_InitSakkeKey_ex**(SakkeKey * key, int keySize, int curveId, void * heap, int devId)
void	**wc_FreeSakkeKey**(SakkeKey * key)
int	**wc_MakeSakkeKey**(SakkeKey * key, WC_RNG * rng)
int	**wc_MakeSakkePublicKey**(SakkeKey * key, ecc_point * pub)
int	**wc_MakeSakkeRsk**(SakkeKey * key, const byte * id, word16 idSz, ecc_point * rsk)
int	**wc_ValidateSakkeRsk**(SakkeKey * key, const byte * id, word16 idSz, ecc_point * rsk, int * valid)
int	**wc_GenerateSakkeRskTable**(const SakkeKey * key, const ecc_point * rsk, byte * table, word32 * len)
int	**wc_ExportSakkeKey**(SakkeKey * key, byte * data, word32 * sz)
int	**wc_ImportSakkeKey**(SakkeKey * key, const byte * data, word32 sz)
int	**wc_ExportSakkePrivateKey**(SakkeKey * key, byte * data, word32 * sz)
int	**wc_ImportSakkePrivateKey**(SakkeKey * key, const byte * data, word32 sz)
int	**wc_EncodeSakkeRsk**(const SakkeKey * key, ecc_point * rsk, byte * out, word32 * sz, int raw)
int	**wc_DecodeSakkeRsk**(const SakkeKey * key, const byte * data, word32 sz, ecc_point * rsk)
int	**wc_ImportSakkeRsk**(SakkeKey * key, const byte * data, word32 sz)
int	**wc_ExportSakkePublicKey**(SakkeKey * key, byte * data, word32 * sz, int raw)
int	**wc_ImportSakkePublicKey**(SakkeKey * key, const byte * data, word32 sz, int trusted)
int	**wc_GetSakkeAuthSize**(SakkeKey * key, word16 * authSz)
int	**wc_SetSakkeIdentity**(SakkeKey * key, const byte * id, word16 idSz)
int	**wc_MakeSakkePointI**(SakkeKey * key, const byte * id, word16 idSz)
int	**wc_GetSakkePointI**(SakkeKey * key, byte * data, word32 * sz)
int	**wc_SetSakkePointI**(SakkeKey * key, const byte * id, word16 idSz, const byte * data, word32 sz)
int	**wc_GenerateSakkePointITable**(SakkeKey * key, byte * table, word32 * len)
int	**wc_SetSakkePointITable**(SakkeKey * key, byte * table, word32 len)
int	**wc_ClearSakkePointITable**(SakkeKey * key)

	Name
int	**wc_MakeSakkeEncapsulatedSSV**(SakkeKey * key, enum wc_HashType hashType, byte * ssv, word16 ssvSz, byte * auth, word16 * authSz)
int	**wc_GenerateSakkeSSV**(SakkeKey * key, WC_RNG * rng, byte * ssv, word16 * ssvSz)
int	**wc_SetSakkeRsk**(SakkeKey * key, const ecc_point * rsk, byte * table, word32 len)
int	**wc_DeriveSakkeSSV**(SakkeKey * key, enum wc_HashType hashType, byte * ssv, word16 ssvSz, const byte * auth, word16 authSz)

C.50.2 Functions Documentation

C.50.2.1 function wc_InitSakkeKey

```
int wc_InitSakkeKey(
    SakkeKey * key,
    void * heap,
    int devId
)
```

C.50.2.2 function wc_InitSakkeKey_ex

```
int wc_InitSakkeKey_ex(
    SakkeKey * key,
    int keySize,
    int curveId,
    void * heap,
    int devId
)
```

C.50.2.3 function wc_FreeSakkeKey

```
void wc_FreeSakkeKey(
    SakkeKey * key
)
```

C.50.2.4 function wc_MakeSakkeKey

```
int wc_MakeSakkeKey(
    SakkeKey * key,
    WC_RNG * rng
)
```

C.50.2.5 function wc_MakeSakkePublicKey

```
int wc_MakeSakkePublicKey(
    SakkeKey * key,
    ecc_point * pub
)
```

C.50.2.6 function wc_MakeSakkeRsk

```
int wc_MakeSakkeRsk(
    SakkeKey * key,
    const byte * id,
    word16 idSz,
    ecc_point * rsk
)
```

C.50.2.7 function wc_ValidateSakkeRsk

```
int wc_ValidateSakkeRsk(
    SakkeKey * key,
    const byte * id,
    word16 idSz,
    ecc_point * rsk,
    int * valid
)
```

C.50.2.8 function wc_GenerateSakkeRskTable

```
int wc_GenerateSakkeRskTable(
    const SakkeKey * key,
    const ecc_point * rsk,
    byte * table,
    word32 * len
)
```

C.50.2.9 function wc_ExportSakkeKey

```
int wc_ExportSakkeKey(
    SakkeKey * key,
    byte * data,
    word32 * sz
)
```

C.50.2.10 function wc_ImportSakkeKey

```
int wc_ImportSakkeKey(
    SakkeKey * key,
    const byte * data,
    word32 sz
)
```

C.50.2.11 function wc_ExportSakkePrivateKey

```
int wc_ExportSakkePrivateKey(
    SakkeKey * key,
    byte * data,
    word32 * sz
)
```

C.50.2.12 function wc_ImportSakkePrivateKey

```
int wc_ImportSakkePrivateKey(
    SakkeKey * key,
    const byte * data,
    word32 sz
)
```

C.50.2.13 function wc_EncodeSakkeRsk

```
int wc_EncodeSakkeRsk(
    const SakkeKey * key,
    ecc_point * rsk,
    byte * out,
    word32 * sz,
    int raw
)
```

C.50.2.14 function wc_DecodeSakkeRsk

```
int wc_DecodeSakkeRsk(
    const SakkeKey * key,
    const byte * data,
    word32 sz,
    ecc_point * rsk
)
```

C.50.2.15 function wc_ImportSakkeRsk

```
int wc_ImportSakkeRsk(
    SakkeKey * key,
    const byte * data,
    word32 sz
)
```

C.50.2.16 function wc_ExportSakkePublicKey

```
int wc_ExportSakkePublicKey(
    SakkeKey * key,
    byte * data,
    word32 * sz,
    int raw
)
```

C.50.2.17 function wc_ImportSakkePublicKey

```
int wc_ImportSakkePublicKey(
    SakkeKey * key,
    const byte * data,
    word32 sz,
    int trusted
)
```

C.50.2.18 function wc_GetSakkeAuthSize

```
int wc_GetSakkeAuthSize(
    SakkeKey * key,
    word16 * authSz
)
```

C.50.2.19 function wc_SetSakkeIdentity

```
int wc_SetSakkeIdentity(
    SakkeKey * key,
    const byte * id,
    word16 idSz
)
```

C.50.2.20 function wc_MakeSakkePointI

```
int wc_MakeSakkePointI(
    SakkeKey * key,
    const byte * id,
    word16 idSz
)
```

C.50.2.21 function wc_GetSakkePointI

```
int wc_GetSakkePointI(
    SakkeKey * key,
    byte * data,
    word32 * sz
)
```

C.50.2.22 function wc_SetSakkePointI

```
int wc_SetSakkePointI(
    SakkeKey * key,
    const byte * id,
    word16 idSz,
    const byte * data,
    word32 sz
)
```

C.50.2.23 function wc_GenerateSakkePointITable

```
int wc_GenerateSakkePointITable(
    SakkeKey * key,
    byte * table,
    word32 * len
)
```

C.50.2.24 function wc_SetSakkePointITable

```
int wc_SetSakkePointITable(
    SakkeKey * key,
    byte * table,
```

```
    word32 len
)
```

C.50.2.25 function wc_ClearSakkePointITable

```
int wc_ClearSakkePointITable(
    SakkeKey * key
)
```

C.50.2.26 function wc_MakeSakkeEncapsulatedSSV

```
int wc_MakeSakkeEncapsulatedSSV(
    SakkeKey * key,
    enum wc_HashType hashType,
    byte * ssv,
    word16 ssvSz,
    byte * auth,
    word16 * authSz
)
```

C.50.2.27 function wc_GenerateSakkeSSV

```
int wc_GenerateSakkeSSV(
    SakkeKey * key,
    WC_RNG * rng,
    byte * ssv,
    word16 * ssvSz
)
```

C.50.2.28 function wc_SetSakkeRsk

```
int wc_SetSakkeRsk(
    SakkeKey * key,
    const ecc_point * rsk,
    byte * table,
    word32 len
)
```

C.50.2.29 function wc_DeriveSakkeSSV

```
int wc_DeriveSakkeSSV(
    SakkeKey * key,
    enum wc_HashType hashType,
    byte * ssv,
    word16 ssvSz,
    const byte * auth,
    word16 authSz
)
```

C.50.3 Source code

```
int wc_InitSakkeKey(SakkeKey* key, void* heap, int devId);
int wc_InitSakkeKey_ex(SakkeKey* key, int keySize, int curveId,
```

```
        void* heap, int devId);
void wc_FreeSakkeKey(SakkeKey* key);

int wc_MakeSakkeKey(SakkeKey* key, WC_RNG* rng);
int wc_MakeSakkePublicKey(SakkeKey* key, ecc_point* pub);

int wc_MakeSakkeRsk(SakkeKey* key, const byte* id, word16 idSz,
        ecc_point* rsk);
int wc_ValidateSakkeRsk(SakkeKey* key, const byte* id, word16 idSz,
        ecc_point* rsk, int* valid);
int wc_GenerateSakkeRskTable(const SakkeKey* key,
        const ecc_point* rsk, byte* table, word32* len);


int wc_ExportSakkeKey(SakkeKey* key, byte* data, word32* sz);
int wc_ImportSakkeKey(SakkeKey* key, const byte* data, word32 sz);
int wc_ExportSakkePrivateKey(SakkeKey* key, byte* data, word32* sz);
int wc_ImportSakkePrivateKey(SakkeKey* key, const byte* data,
        word32 sz);

int wc_EncodeSakkeRsk(const SakkeKey* key, ecc_point* rsk,
        byte* out, word32* sz, int raw);
int wc_DecodeSakkeRsk(const SakkeKey* key, const byte* data,
        word32 sz, ecc_point* rsk);

int wc_ImportSakkeRsk(SakkeKey* key, const byte* data, word32 sz);

int wc_ExportSakkePublicKey(SakkeKey* key, byte* data,
        word32* sz, int raw);
int wc_ImportSakkePublicKey(SakkeKey* key, const byte* data,
        word32 sz, int trusted);

int wc_GetSakkeAuthSize(SakkeKey* key, word16* authSz);
int wc_SetSakkeIdentity(SakkeKey* key, const byte* id, word16 idSz);
int wc_MakeSakkePointI(SakkeKey* key, const byte* id, word16 idSz);
int wc_GetSakkePointI(SakkeKey* key, byte* data, word32* sz);
int wc_SetSakkePointI(SakkeKey* key, const byte* id, word16 idSz,
        const byte* data, word32 sz);
int wc_GenerateSakkePointITable(SakkeKey* key, byte* table,
         word32* len);
int wc_SetSakkePointITable(SakkeKey* key, byte* table, word32 len);
int wc_ClearSakkePointITable(SakkeKey* key);
int wc_MakeSakkeEncapsulatedSSV(SakkeKey* key,
        enum wc_HashType hashType, byte* ssv, word16 ssvSz, byte* auth,
        word16* authSz);
int wc_GenerateSakkeSSV(SakkeKey* key, WC_RNG* rng, byte* ssv,
        word16* ssvSz);
int wc_SetSakkeRsk(SakkeKey* key, const ecc_point* rsk, byte* table,
        word32 len);
int wc_DeriveSakkeSSV(SakkeKey* key, enum wc_HashType hashType,
        byte* ssv, word16 ssvSz, const byte* auth,
        word16 authSz);
```

C.51 sha256.h

C.51.1 Functions

	Name
int	**wc_InitSha256**(wc_Sha256 * sha)This function initializes SHA256. This is automatically called by wc_Sha256Hash.
int	**wc_Sha256Update**(wc_Sha256 * sha, const byte * data, word32 len)Can be called to continually hash the provided byte array of length len.
int	**wc_Sha256Final**(wc_Sha256 * sha256, byte * hash)Finalizes hashing of data. Result is placed into hash. Resets state of sha256 struct.
void	**wc_Sha256Free**(wc_Sha256 * sha256)Resets the Sha256 structure. Note: this is only supported if you have WOLFSSL_TI_HASH defined.
int	**wc_Sha256Reset**(wc_Sha256 * sha256)Resets the sha256 structure to its freshly initialized state, continuing with its established allocations, heap hint, and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha256Free for teardown at end of life.
int	**wc_Sha256GetHash**(wc_Sha256 * sha256, byte * hash)Gets hash data. Result is placed into hash. Does not reset state of sha256 struct.
int	**wc_InitSha224**(wc_Sha224 * sha224)Used to initialize a Sha224 struct.
int	**wc_Sha224Update**(wc_Sha224 * sha224, const byte * data, word32 len)Can be called to continually hash the provided byte array of length len.
int	**wc_Sha224Final**(wc_Sha224 * sha224, byte * hash)Finalizes hashing of data. Result is placed into hash. Resets state of sha224 struct.
int	**wc_InitSha256_ex**(wc_Sha256 * sha, void * heap, int devId)Initializes SHA256 with heap and device ID.
int	**wc_Sha256FinalRaw**(wc_Sha256 * sha256, byte * hash)Gets raw hash without finalizing.
int	**wc_Sha256Transform**(wc_Sha256 * sha, const unsigned char * data)Transforms SHA256 block.
int	**wc_Sha256HashBlock**(wc_Sha256 * sha, const unsigned char * data, unsigned char * hash)Hashes single block and outputs result.

	Name
int	**wc_Sha256_Grow**(wc_Sha256 * sha256, const byte * in, int inSz)Grows SHA256 buffer with input data. This function is only available when WOLFSSL_HASH_KEEP is defined. It is used for keeping an internal buffer to hold all data to be hashed rather than iterating over update, which is necessary for some hardware acceleration platforms that have restrictions on streaming hash operations.
int	**wc_Sha256Copy**(wc_Sha256 * src, wc_Sha256 * dst)Copies SHA256 context.
void	**wc_Sha256SizeSet**(wc_Sha256 * sha256, word32 len)Sets SHA256 size.
int	**wc_Sha256SetFlags**(wc_Sha256 * sha256, word32 flags)Sets SHA256 flags.
int	**wc_Sha256GetFlags**(wc_Sha256 * sha256, word32 * flags)Gets SHA256 flags.
int	**wc_InitSha224_ex**(wc_Sha224 * sha224, void * heap, int devId)Initializes SHA224 with heap and device ID.
void	**wc_Sha224Free**(wc_Sha224 * sha224)Frees SHA224 resources.
int	**wc_Sha224Reset**(wc_Sha224 * sha224)Resets the sha224 structure to its freshly initialized state, continuing with its established allocations, heap hint, and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha224Free for teardown at end of life.
int	**wc_Sha224_Grow**(wc_Sha224 * sha224, const byte * in, int inSz)Grows SHA224 buffer with input data. This function is only available when WOLFSSL_HASH_KEEP is defined. It is used for keeping an internal buffer to hold all data to be hashed rather than iterating over update, which is necessary for some hardware acceleration platforms that have restrictions on streaming hash operations.
int	**wc_Sha224GetHash**(wc_Sha224 * sha224, byte * hash)Gets SHA224 hash without finalizing.
int	**wc_Sha224Copy**(wc_Sha224 * src, wc_Sha224 * dst)Copies SHA224 context.
int	**wc_Sha224SetFlags**(wc_Sha224 * sha224, word32 flags)Sets SHA224 flags.
int	**wc_Sha224GetFlags**(wc_Sha224 * sha224, word32 * flags)Gets SHA224 flags.

C.51.2 Functions Documentation

C.51.2.1 function wc_InitSha256

```
int wc_InitSha256(
    wc_Sha256 * sha
)
```

This function initializes SHA256. This is automatically called by wc_Sha256Hash.

Parameters:

- **sha256** pointer to the sha256 structure to use for encryption

See:

- wc_Sha256Hash
- wc_Sha256Update
- wc_Sha256Final

Return: 0 Returned upon successfully initializing

Example

```
Sha256 sha256[1];
if ((ret = wc_InitSha256(sha256)) != 0) {
    WOLFSSL_MSG("wc_InitSha256 failed");
}
else {
    wc_Sha256Update(sha256, data, len);
    wc_Sha256Final(sha256, hash);
}
```

C.51.2.2 function wc_Sha256Update

```
int wc_Sha256Update(
    wc_Sha256 * sha,
    const byte * data,
    word32 len
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **sha256** pointer to the sha256 structure to use for encryption
- **data** the data to be hashed
- **len** length of data to be hashed

See:

- wc_Sha256Hash
- wc_Sha256Final
- wc_InitSha256

Return: 0 Returned upon successfully adding the data to the digest.

Example

```
Sha256 sha256[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);
```

```
if ((ret = wc_InitSha256(sha256)) != 0) {
   WOLFSSL_MSG("wc_InitSha256 failed");
}
else {
    wc_Sha256Update(sha256, data, len);
    wc_Sha256Final(sha256, hash);
}
```

C.51.2.3 function wc_Sha256Final

```
int wc_Sha256Final(
    wc_Sha256 * sha256,
    byte * hash
)
```

Finalizes hashing of data. Result is placed into hash. Resets state of sha256 struct.

Parameters:

- **sha256** pointer to the sha256 structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Sha256Hash
- wc_Sha256GetHash
- wc_InitSha256

Return: 0 Returned upon successfully finalizing.

Example

```
Sha256 sha256[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha256(sha256)) != 0) {
   WOLFSSL_MSG("wc_InitSha256 failed");
}
else {
   wc_Sha256Update(sha256, data, len);
   wc_Sha256Final(sha256, hash);
}
```

C.51.2.4 function wc_Sha256Free

```
void wc_Sha256Free(
    wc_Sha256 * sha256
)
```

Resets the Sha256 structure. Note: this is only supported if you have WOLFSSL_TI_HASH defined.

Parameters:

- **sha256** Pointer to the sha256 structure to be freed.

See:

- wc_InitSha256
- wc_Sha256Update
- wc_Sha256Final

Return: none No returns.

Example

```
Sha256 sha256;
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha256(&sha256)) != 0) {
    WOLFSSL_MSG("wc_InitSha256 failed");
}
else {
    wc_Sha256Update(&sha256, data, len);
    wc_Sha256Final(&sha256, hash);
    wc_Sha256Free(&sha256);
}
```

C.51.2.5 function wc_Sha256Reset

```
int wc_Sha256Reset(
    wc_Sha256 * sha256
)
```

Resets the sha256 structure to its freshly initialized state, continuing with its established allocations, heap hint, and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha256Free for teardown at end of life.

Parameters:

- **sha256** Pointer to the sha256 structure to be reset.

See:

- wc_InitSha256
- wc_Sha256Final
- wc_Sha256Free

Return:

- 0 Returned upon successfully resetting the structure.
- BAD_FUNC_ARG Returned if sha256 is NULL.
- negative Other values may be returned on reinitialization failure in fallback (non-plain-software) configurations.

Example

```
wc_Sha256 sha256;
byte data[] = { Data to be discarded };
byte data2[] = { Data to be hashed };
byte hash[WC_SHA256_DIGEST_SIZE];

if ((ret = wc_InitSha256(&sha256)) == 0) {
    wc_Sha256Update(&sha256, data, sizeof(data));
    wc_Sha256Reset(&sha256); /* discard the partially hashed message */
    wc_Sha256Update(&sha256, data2, sizeof(data2));
    wc_Sha256Final(&sha256, hash);
    wc_Sha256Free(&sha256);
}
```

C.51.2.6 function wc_Sha256GetHash

```
int wc_Sha256GetHash(
    wc_Sha256 * sha256,
    byte * hash
)
```

Gets hash data. Result is placed into hash. Does not reset state of sha256 struct.

Parameters:

- **sha256** pointer to the sha256 structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Sha256Hash
- wc_Sha256Final
- wc_InitSha256

Return: 0 Returned upon successfully finalizing.

Example

```
Sha256 sha256[1];
if ((ret = wc_InitSha256(sha256)) != 0) {
   WOLFSSL_MSG("wc_InitSha256 failed");
}
else {
   wc_Sha256Update(sha256, data, len);
   wc_Sha256GetHash(sha256, hash);
}
```

C.51.2.7 function wc_InitSha224

```
int wc_InitSha224(
    wc_Sha224 * sha224
)
```

Used to initialize a Sha224 struct.

Parameters:

- **sha224** Pointer to a Sha224 struct to initialize.

See:

- wc_Sha224Hash
- wc_Sha224Update
- wc_Sha224Final

Return:

- 0 Success
- 1 Error returned because sha224 is null.

Example

```
Sha224 sha224;
if(wc_InitSha224(&sha224) != 0)
{
    // Handle error
}
```

C.51.2.8 function wc_Sha224Update

```
int wc_Sha224Update(
    wc_Sha224 * sha224,
    const byte * data,
    word32 len
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **sha224** Pointer to the Sha224 structure to use for encryption.
- **data** Data to be hashed.
- **len** Length of data to be hashed.

See:

- wc_InitSha224
- wc_Sha224Final
- wc_Sha224Hash

Return:

- 0 Success
- 1 Error returned if function fails.
- BAD_FUNC_ARG Error returned if sha224 or data is null.

Example

```
Sha224 sha224;
byte data[]; // Data to be hashed
word32 len = sizeof(data);

if ((ret = wc_InitSha224(&sha224)) != 0) {
   WOLFSSL_MSG("wc_InitSha224 failed");
}
else {
  wc_Sha224Update(&sha224, data, len);
  wc_Sha224Final(&sha224, hash);
}
```

C.51.2.9 function wc_Sha224Final

```
int wc_Sha224Final(
    wc_Sha224 * sha224,
    byte * hash
)
```

Finalizes hashing of data. Result is placed into hash. Resets state of sha224 struct.

Parameters:

- **sha224** pointer to the sha224 structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_InitSha224
- wc_Sha224Hash
- wc_Sha224Update

Return:

- 0 Success
- <0 Error

Example

```
Sha224 sha224;
byte data[]; // Data to be hashed
word32 len = sizeof(data);

if ((ret = wc_InitSha224(&sha224)) != 0) {
    WOLFSSL_MSG("wc_InitSha224 failed");
}
else {
    wc_Sha224Update(&sha224, data, len);
    wc_Sha224Final(&sha224, hash);
}
```

C.51.2.10 function wc_InitSha256_ex

```
int wc_InitSha256_ex(
    wc_Sha256 * sha,
    void * heap,
    int devId
)
```

Initializes SHA256 with heap and device ID.

Parameters:

- **sha** SHA256 structure
- **heap** Heap hint
- **devId** Device ID

See: wc_InitSha256

Return:

- 0 on success
- negative on error

Example

```
wc_Sha256 sha;
int ret = wc_InitSha256_ex(&sha, NULL, INVALID_DEVID);
```

C.51.2.11 function wc_Sha256FinalRaw

```
int wc_Sha256FinalRaw(
    wc_Sha256 * sha256,
    byte * hash
)
```

Gets raw hash without finalizing.

Parameters:

- **sha256** SHA256 structure
- **hash** Output hash buffer

See: wc_Sha256Final

Return:

- 0 on success
- negative on error

Example

```
wc_Sha256 sha;
byte hash[WC_SHA256_DIGEST_SIZE];
int ret = wc_Sha256FinalRaw(&sha, hash);
```

C.51.2.12 function wc_Sha256Transform

```
int wc_Sha256Transform(
    wc_Sha256 * sha,
    const unsigned char * data
)
```

Transforms SHA256 block.

Parameters:

- **sha** SHA256 structure
- **data** Block data

See: wc_Sha256Update

Return:

- 0 on success
- negative on error

Example

```
wc_Sha256 sha;
unsigned char block[WC_SHA256_BLOCK_SIZE];
int ret = wc_Sha256Transform(&sha, block);
```

C.51.2.13 function wc_Sha256HashBlock

```
int wc_Sha256HashBlock(
    wc_Sha256 * sha,
    const unsigned char * data,
    unsigned char * hash
)
```

Hashes single block and outputs result.

Parameters:

- **sha** SHA256 structure
- **data** Block data
- **hash** Output hash buffer

See: wc_Sha256Transform

Return:

- 0 on success
- negative on error

Example

```
wc_Sha256 sha;
unsigned char block[WC_SHA256_BLOCK_SIZE];
unsigned char hash[WC_SHA256_DIGEST_SIZE];
int ret = wc_Sha256HashBlock(&sha, block, hash);
```

C.51.2.14 function wc_Sha256_Grow

```
int wc_Sha256_Grow(
    wc_Sha256 * sha256,
    const byte * in,
    int inSz
)
```

Grows SHA256 buffer with input data. This function is only available when WOLFSSL_HASH_KEEP is defined. It is used for keeping an internal buffer to hold all data to be hashed rather than iterating over update, which is necessary for some hardware acceleration platforms that have restrictions on streaming hash operations.

Parameters:

- **sha256** SHA256 structure
- **in** Input data
- **inSz** Input size

See: wc_Sha256Update

Return:

- 0 on success
- negative on error

Example

```
wc_Sha256 sha;
byte data[100];
int ret = wc_Sha256_Grow(&sha, data, sizeof(data));
```

C.51.2.15 function wc_Sha256Copy

```
int wc_Sha256Copy(
    wc_Sha256 * src,
    wc_Sha256 * dst
)
```

Copies SHA256 context.

Parameters:

- **src** Source SHA256 structure
- **dst** Destination SHA256 structure; (must be zeroed or previously initialized)

See: wc_InitSha256

Return:

- 0 on success
- negative on error

Example

```
wc_Sha256 src, dst;
memset(&dst, 0, sizeof(dst));
```

```
int ret = wc_InitSha256(&src);
if (ret == 0)
    ret = wc_Sha256Copy(&src, &dst);
```

C.51.2.16 function wc_Sha256SizeSet

```
void wc_Sha256SizeSet(
    wc_Sha256 * sha256,
    word32 len
)
```

Sets SHA256 size.

Parameters:

- **sha256** SHA256 structure
- **len** Size to set

See: wc_Sha256Update

Return: none No returns

Example

```
wc_Sha256 sha;
wc_Sha256SizeSet(&sha, 1000);
```

C.51.2.17 function wc_Sha256SetFlags

```
int wc_Sha256SetFlags(
    wc_Sha256 * sha256,
    word32 flags
)
```

Sets SHA256 flags.

Parameters:

- **sha256** SHA256 structure
- **flags** Flags to set

See: wc_InitSha256

Return:

- 0 on success
- negative on error

Example

```
wc_Sha256 sha;
int ret = wc_Sha256SetFlags(&sha, WC_HASH_FLAG_WILLCOPY);
```

C.51.2.18 function wc_Sha256GetFlags

```
int wc_Sha256GetFlags(
    wc_Sha256 * sha256,
    word32 * flags
)
```

Gets SHA256 flags.

Parameters:

- **sha256** SHA256 structure
- **flags** Pointer to store flags

See: wc_Sha256SetFlags

Return:

- 0 on success
- negative on error

Example

```
wc_Sha256 sha;
word32 flags;
int ret = wc_Sha256GetFlags(&sha, &flags);
```

C.51.2.19 function wc_InitSha224_ex

```
int wc_InitSha224_ex(
    wc_Sha224 * sha224,
    void * heap,
    int devId
)
```

Initializes SHA224 with heap and device ID.

Parameters:

- **sha224** SHA224 structure
- **heap** Heap hint
- **devId** Device ID

See: wc_InitSha224

Return:

- 0 on success
- negative on error

Example

```
wc_Sha224 sha;
int ret = wc_InitSha224_ex(&sha, NULL, INVALID_DEVID);
```

C.51.2.20 function wc_Sha224Free

```
void wc_Sha224Free(
    wc_Sha224 * sha224
)
```

Frees SHA224 resources.

Parameters:

- **sha224** SHA224 structure

See: wc_InitSha224

Return: none No returns

Example

```
wc_Sha224 sha;
wc_InitSha224(&sha);
wc_Sha224Free(&sha);
```

C.51.2.21 function wc_Sha224Reset

```
int wc_Sha224Reset(
    wc_Sha224 * sha224
)
```

Resets the sha224 structure to its freshly initialized state, continuing with its established allocations, heap hint, and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha224Free for teardown at end of life.

Parameters:

- **sha224** Pointer to the sha224 structure to be reset.

See:

- wc_InitSha224
- wc_Sha224Final
- wc_Sha224Free

Return:

- 0 Returned upon successfully resetting the structure.
- BAD_FUNC_ARG Returned if sha224 is NULL.
- negative Other values may be returned on reinitialization failure in fallback (non-plain-software) configurations.

Example

```
wc_Sha224 sha224;
byte data[] = { Data to be discarded };
byte data2[] = { Data to be hashed };
byte hash[WC_SHA224_DIGEST_SIZE];

if ((ret = wc_InitSha224(&sha224)) == 0) {
    wc_Sha224Update(&sha224, data, sizeof(data));
    wc_Sha224Reset(&sha224); /* discard the partially hashed message */
    wc_Sha224Update(&sha224, data2, sizeof(data2));
    wc_Sha224Final(&sha224, hash);
    wc_Sha224Free(&sha224);
}
```

C.51.2.22 function wc_Sha224_Grow

```
int wc_Sha224_Grow(
    wc_Sha224 * sha224,
    const byte * in,
    int inSz
)
```

Grows SHA224 buffer with input data. This function is only available when WOLFSSL_HASH_KEEP is defined. It is used for keeping an internal buffer to hold all data to be hashed rather than iterating

over update, which is necessary for some hardware acceleration platforms that have restrictions on streaming hash operations.

Parameters:

- **sha224** SHA224 structure
- **in** Input data
- **inSz** Input size

See: wc_Sha224Update

Return:

- 0 on success
- negative on error

Example

```
wc_Sha224 sha;
byte data[100];
int ret = wc_Sha224_Grow(&sha, data, sizeof(data));
```

C.51.2.23 function wc_Sha224GetHash

```
int wc_Sha224GetHash(
    wc_Sha224 * sha224,
    byte * hash
)
```

Gets SHA224 hash without finalizing.

Parameters:

- **sha224** SHA224 structure
- **hash** Output hash buffer

See: wc_Sha224Final

Return:

- 0 on success
- negative on error

Example

```
wc_Sha224 sha;
byte hash[WC_SHA224_DIGEST_SIZE];
int ret = wc_Sha224GetHash(&sha, hash);
```

C.51.2.24 function wc_Sha224Copy

```
int wc_Sha224Copy(
    wc_Sha224 * src,
    wc_Sha224 * dst
)
```

Copies SHA224 context.

Parameters:

- **src** Source SHA224 structure
- **dst** Destination SHA224 structure (must be zeroed or previously initialized)

See: wc_InitSha224

Return:

- 0 on success
- negative on error

Example

```
wc_Sha224 src, dst;
memset(&dst, 0, sizeof(dst));
int ret = wc_InitSha224(&src);
if (ret == 0)
    ret = wc_Sha224Copy(&src, &dst);
```

C.51.2.25 function wc_Sha224SetFlags

```
int wc_Sha224SetFlags(
    wc_Sha224 * sha224,
    word32 flags
)
```

Sets SHA224 flags.

Parameters:

- **sha224** SHA224 structure
- **flags** Flags to set

See: wc_InitSha224

Return:

- 0 on success
- negative on error

Example

```
wc_Sha224 sha;
int ret = wc_Sha224SetFlags(&sha, WC_HASH_FLAG_WILLCOPY);
```

C.51.2.26 function wc_Sha224GetFlags

```
int wc_Sha224GetFlags(
    wc_Sha224 * sha224,
    word32 * flags
)
```

Gets SHA224 flags.

Parameters:

- **sha224** SHA224 structure
- **flags** Pointer to store flags

See: wc_Sha224SetFlags

Return:

- 0 on success
- negative on error

Example

```
wc_Sha224 sha;
word32 flags;
int ret = wc_Sha224GetFlags(&sha, &flags);
```

C.51.3 Source code

```
int wc_InitSha256(wc_Sha256* sha);

int wc_Sha256Update(wc_Sha256* sha, const byte* data, word32 len);

int wc_Sha256Final(wc_Sha256* sha256, byte* hash);

void wc_Sha256Free(wc_Sha256* sha256);

        wc_Sha256Update(&sha256, data2, sizeof(data2));
        wc_Sha256Final(&sha256, hash);
        wc_Sha256Free(&sha256);
    }
    \endcode

    \sa wc_InitSha256
    \sa wc_Sha256Final
    \sa wc_Sha256Free
*/
int wc_Sha256Reset(wc_Sha256* sha256);

int wc_Sha256GetHash(wc_Sha256* sha256, byte* hash);

int wc_InitSha224(wc_Sha224* sha224);

int wc_Sha224Update(wc_Sha224* sha224, const byte* data, word32 len);

int wc_Sha224Final(wc_Sha224* sha224, byte* hash);

int wc_InitSha256_ex(wc_Sha256* sha, void* heap, int devId);

int wc_Sha256FinalRaw(wc_Sha256* sha256, byte* hash);

int wc_Sha256Transform(wc_Sha256* sha, const unsigned char* data);

int wc_Sha256HashBlock(wc_Sha256* sha, const unsigned char* data,
    unsigned char* hash);

int wc_Sha256_Grow(wc_Sha256* sha256, const byte* in, int inSz);

int wc_Sha256Copy(wc_Sha256* src, wc_Sha256* dst);

void wc_Sha256SizeSet(wc_Sha256* sha256, word32 len);

int wc_Sha256SetFlags(wc_Sha256* sha256, word32 flags);

int wc_Sha256GetFlags(wc_Sha256* sha256, word32* flags);
```

```
int wc_InitSha224_ex(wc_Sha224* sha224, void* heap, int devId);

void wc_Sha224Free(wc_Sha224* sha224);

        wc_Sha224Update(&sha224, data2, sizeof(data2));
        wc_Sha224Final(&sha224, hash);
        wc_Sha224Free(&sha224);
    }
    \endcode

    \sa wc_InitSha224
    \sa wc_Sha224Final
    \sa wc_Sha224Free
*/
int wc_Sha224Reset(wc_Sha224* sha224);

int wc_Sha224_Grow(wc_Sha224* sha224, const byte* in, int inSz);

int wc_Sha224GetHash(wc_Sha224* sha224, byte* hash);

int wc_Sha224Copy(wc_Sha224* src, wc_Sha224* dst);

int wc_Sha224SetFlags(wc_Sha224* sha224, word32 flags);

int wc_Sha224GetFlags(wc_Sha224* sha224, word32* flags);
```

C.52 sha3.h

C.52.1 Functions

	Name
int	**wc_InitSha3_224**(wc_Sha3 * sha3, void * heap, int devId)This function initializes SHA3-224. This is automatically called by wc_Sha3_224Hash.
int	**wc_Sha3_224_Update**(wc_Sha3 * sha, const byte * data, word32 len)Can be called to continually hash the provided byte array of length len.
int	**wc_Sha3_224_Final**(wc_Sha3 * sha3, byte * hash)Finalizes hashing of data. Result is placed into hash. Resets state of sha3 struct.
void	**wc_Sha3_224_Free**(wc_Sha3 * sha3)Resets the wc_Sha3 structure. Note: this is only supported if you have WOLFSSL_TI_HASH defined.

	Name
int	**wc_Sha3_224_Reset**(wc_Sha3 * sha3)Resets the sha3 structure to its freshly initialized state, continuing with the established heap hint and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha3_224_Free for teardown at end of life.
int	**wc_Sha3_224_GetHash**(wc_Sha3 * sha3, byte * hash)Gets hash data. Result is placed into hash. Does not reset state of sha3 struct.
int	**wc_Sha3_224_Copy**(wc_Sha3 * sha3, wc_Sha3 * dst)Copy the state of the hash.
int	**wc_InitSha3_256**(wc_Sha3 * sha3, void * heap, int devId)This function initializes SHA3-256. This is automatically called by wc_Sha3_256Hash.
int	**wc_Sha3_256_Update**(wc_Sha3 * sha, const byte * data, word32 len)Can be called to continually hash the provided byte array of length len.
int	**wc_Sha3_256_Final**(wc_Sha3 * sha3, byte * hash)Finalizes hashing of data. Result is placed into hash. Resets state of sha3 struct.
void	**wc_Sha3_256_Free**(wc_Sha3 * sha3)Resets the wc_Sha3 structure. Note: this is only supported if you have WOLFSSL_TI_HASH defined.
int	**wc_Sha3_256_Reset**(wc_Sha3 * sha3)Resets the sha3 structure to its freshly initialized state, continuing with the established heap hint and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha3_256_Free for teardown at end of life.
int	**wc_Sha3_256_GetHash**(wc_Sha3 * sha3, byte * hash)Gets hash data. Result is placed into hash. Does not reset state of sha3 struct.
int	**wc_Sha3_256_Copy**(wc_Sha3 * sha3, wc_Sha3 * dst)Copy the state of the hash.
int	**wc_InitSha3_384**(wc_Sha3 * sha3, void * heap, int devId)This function initializes SHA3-384. This is automatically called by wc_Sha3_384Hash.
int	**wc_Sha3_384_Update**(wc_Sha3 * sha, const byte * data, word32 len)Can be called to continually hash the provided byte array of length len.

	Name
int	**wc_Sha3_384_Final**(wc_Sha3 * sha3, byte * hash)Finalizes hashing of data. Result is placed into hash. Resets state of sha3 struct.
void	**wc_Sha3_384_Free**(wc_Sha3 * sha3)Resets the wc_Sha3 structure. Note: this is only supported if you have WOLFSSL_TI_HASH defined.
int	**wc_Sha3_384_Reset**(wc_Sha3 * sha3)Resets the sha3 structure to its freshly initialized state, continuing with the established heap hint and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha3_384_Free for teardown at end of life.
int	**wc_Sha3_384_GetHash**(wc_Sha3 * sha3, byte * hash)Gets hash data. Result is placed into hash. Does not reset state of sha3 struct.
int	**wc_Sha3_384_Copy**(wc_Sha3 * sha3, wc_Sha3 * dst)Copy the state of the hash.
int	**wc_InitSha3_512**(wc_Sha3 * sha3, void * heap, int devId)This function initializes SHA3-512. This is automatically called by wc_Sha3_512Hash.
int	**wc_Sha3_512_Update**(wc_Sha3 * sha, const byte * data, word32 len)Can be called to continually hash the provided byte array of length len.
int	**wc_Sha3_512_Final**(wc_Sha3 * sha3, byte * hash)Finalizes hashing of data. Result is placed into hash. Resets state of sha3 struct.
void	**wc_Sha3_512_Free**(wc_Sha3 * sha3)Resets the wc_Sha3 structure. Note: this is only supported if you have WOLFSSL_TI_HASH defined.
int	**wc_Sha3_512_Reset**(wc_Sha3 * sha3)Resets the sha3 structure to its freshly initialized state, continuing with the established heap hint and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha3_512_Free for teardown at end of life.
int	**wc_Sha3_512_GetHash**(wc_Sha3 * sha3, byte * hash)Gets hash data. Result is placed into hash. Does not reset state of sha3 struct.
int	**wc_Sha3_512_Copy**(wc_Sha3 * sha3, wc_Sha3 * dst)Copy the state of the hash.

	Name
int	**wc_InitShake128**(wc_Shake * shake, void * heap, int devId)This function initializes SHAKE-128. This is automatically called by wc_Shake128Hash.
int	**wc_Shake128_Update**(wc_Shake * sha, const byte * data, word32 len)Can be called to continually hash the provided byte array of length len.
int	**wc_Shake128_Final**(wc_Shake * shake, byte * hash, word32 hashLen)Finalizes hashing of data. Result is placed into hash. Resets state of shake struct.
int	**wc_Shake128_Absorb**(wc_Shake * sha, const byte * data, word32 len)Called to absorb the provided byte array of length len. Cannot be called incrementally.
int	**wc_Shake128_SqueezeBlocks**(wc_Shake * shake, byte * out, word32 blockCnt)Squeeze out more blocks of data. Result is placed into out. Can be called inrementally.
void	**wc_Shake128_Free**(wc_Shake * shake)Resets the wc_Shake structure. Note: this is only supported if you have WOLFSSL_TI_HASH defined.
int	**wc_Shake128_Reset**(wc_Shake * shake)Resets the shake structure to its freshly initialized state, continuing with the established heap hint and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially absorbed input, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Shake128_Free for teardown at end of life.
int	**wc_Shake128_GetHash**(wc_Shake * shake, byte * hash)Gets hash data. Result is placed into hash. Does not reset state of shake struct.
int	**wc_Shake128_Copy**(wc_Shake * src, wc_Sha3 * dst)Copy the state of the hash.
int	**wc_InitShake256**(wc_Shake * shake, void * heap, int devId)This function initializes SHAKE-256. This is automatically called by wc_Shake256Hash.
int	**wc_Shake256_Update**(wc_Shake * sha, const byte * data, word32 len)Can be called to continually hash the provided byte array of length len.
int	**wc_Shake256_Final**(wc_Shake * shake, byte * hash, word32 hashLen)Finalizes hashing of data. Result is placed into hash. Resets state of shake struct.

	Name
int	**wc_Shake256_Absorb**(wc_Shake * sha, const byte * data, word32 len)Called to absorb the provided byte array of length len. Cannot be called incrementally.
int	**wc_Shake256_SqueezeBlocks**(wc_Shake * shake, byte * out, word32 blockCnt)Squeeze out more blocks of data. Result is placed into out. Can be called incrementally.
void	**wc_Shake256_Free**(wc_Shake * shake)Resets the wc_Shake structure. Note: this is only supported if you have WOLFSSL_TI_HASH defined.
int	**wc_Shake256_Reset**(wc_Shake * shake)Resets the shake structure to its freshly initialized state, continuing with the established heap hint and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially absorbed input, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Shake256_Free for teardown at end of life.
int	**wc_Shake256_GetHash**(wc_Shake * shake, byte * hash)Gets hash data. Result is placed into hash. Does not reset state of shake struct.
int	**wc_Shake256_Copy**(wc_Shake * src, wc_Sha3 * dst)Copy the state of the hash.
int	**wc_InitKmac128**(wc_Kmac * kmac, const byte * key, word32 keyLen, const byte * custom, word32 customLen, void * heap, int devId)This function initializes a KMAC128 operation with a key and an optional customization string, per NIST SP 800_185. KMAC128 requires SHAKE128 (WOLFSSL_SHAKE128) and is enabled with WOLFSSL_KMAC.
int	**wc_Kmac128_Update**(wc_Kmac * kmac, const byte * in, word32 inLen)This function absorbs message data into a KMAC128 operation. It may be called any number of times before finalizing.
int	**wc_Kmac128_Final**(wc_Kmac * kmac, byte * out, word32 outLen)This function finalizes a KMAC128 operation, producing outLen bytes of output. The output length is bound into the result, so a KMAC with a different length is a different value.
int	**wc_Kmac128_FinalXof**(wc_Kmac * kmac, byte * out, word32 outLen)This function finalizes a KMAC128 operation as an XOF (KMACXOF128). Unlike wc_Kmac128_Final, the output length is not bound into the result, so any amount of output may be requested.

	Name
void	**wc_Kmac128_Free**(wc_Kmac * kmac)This function releases any resources associated with a KMAC128 operation. Passing NULL is safe and does nothing.
int	**wc_Kmac128Hash**(const byte * key, word32 keyLen, const byte * custom, word32 customLen, const byte * in, word32 inLen, byte * out, word32 outLen)This function computes a KMAC128 over a single message in one call, initializing, updating and finalizing internally.
int	**wc_Kmac128HashXof**(const byte * key, word32 keyLen, const byte * custom, word32 customLen, const byte * in, word32 inLen, byte * out, word32 outLen)This function computes a KMACXOF128 over a single message in one call. Like wc_Kmac128Hash but the output length is not bound into the result, so any amount of output may be requested.
int	**wc_InitKmac256**(wc_Kmac * kmac, const byte * key, word32 keyLen, const byte * custom, word32 customLen, void * heap, int devId)This function initializes a KMAC256 operation with a key and an optional customization string, per NIST SP 800_185. KMAC256 requires SHAKE256 (WOLFSSL_SHAKE256) and is enabled with WOLFSSL_KMAC.
int	**wc_Kmac256_Update**(wc_Kmac * kmac, const byte * in, word32 inLen)This function absorbs message data into a KMAC256 operation. It may be called any number of times before finalizing.
int	**wc_Kmac256_Final**(wc_Kmac * kmac, byte * out, word32 outLen)This function finalizes a KMAC256 operation, producing outLen bytes of output. The output length is bound into the result, so a KMAC with a different length is a different value.
int	**wc_Kmac256_FinalXof**(wc_Kmac * kmac, byte * out, word32 outLen)This function finalizes a KMAC256 operation as an XOF (KMACXOF256). Unlike wc_Kmac256_Final, the output length is not bound into the result, so any amount of output may be requested.
void	**wc_Kmac256_Free**(wc_Kmac * kmac)This function releases any resources associated with a KMAC256 operation. Passing NULL is safe and does nothing.

	Name
int	**wc_Kmac256Hash**(const byte * key, word32 keyLen, const byte * custom, word32 customLen, const byte * in, word32 inLen, byte * out, word32 outLen)This function computes a KMAC256 over a single message in one call, initializing, updating and finalizing internally.
int	**wc_Kmac256HashXof**(const byte * key, word32 keyLen, const byte * custom, word32 customLen, const byte * in, word32 inLen, byte * out, word32 outLen)This function computes a KMACXOF256 over a single message in one call. Like wc_Kmac256Hash but the output length is not bound into the result, so any amount of output may be requested.
int	**wc_Kmac128_Copy**(wc_Kmac * src, wc_Kmac * dst)This function copies the state of a KMAC128 operation so it can be finalized more than once, for example over a common message prefix.
int	**wc_Kmac256_Copy**(wc_Kmac * src, wc_Kmac * dst)This function copies the state of a KMAC256 operation so it can be finalized more than once, for example over a common message prefix.
int	**wc_InitCshake128**(wc_Cshake * cshake, const byte * name, word32 nameLen, const byte * custom, word32 customLen, void * heap, int devId)This function initializes a cSHAKE128 operation (customizable SHAKE, NIST SP 800_185) with a function_name and customization string. cSHAKE is enabled together with KMAC (WOLFSSL_KMAC). When both strings are empty, cSHAKE reduces to plain SHAKE128.
int	**wc_Cshake128_Update**(wc_Cshake * cshake, const byte * in, word32 inLen)This function absorbs message data into a cSHAKE128 operation.
int	**wc_Cshake128_Final**(wc_Cshake * cshake, byte * out, word32 outLen)This function finalizes a cSHAKE128 operation, squeezing outLen bytes of output. cSHAKE is an XOF, so a longer squeeze extends a shorter one over the same input.
void	**wc_Cshake128_Free**(wc_Cshake * cshake)This function releases any resources associated with a cSHAKE128 operation. Passing NULL is safe and does nothing.
int	**wc_Cshake128**(const byte * name, word32 nameLen, const byte * custom, word32 customLen, const byte * in, word32 inLen, byte * out, word32 outLen)This function computes a cSHAKE128 over a single message in one call, initializing, updating and finalizing internally.

	Name
int	**wc_InitCshake256**(wc_Cshake * cshake, const byte * name, word32 nameLen, const byte * custom, word32 customLen, void * heap, int devId)This function initializes a cSHAKE256 operation (customizable SHAKE, NIST SP 800_185) with a function_name and customization string. cSHAKE is enabled together with KMAC (WOLFSSL_KMAC). When both strings are empty, cSHAKE reduces to plain SHAKE256.
int	**wc_Cshake256_Update**(wc_Cshake * cshake, const byte * in, word32 inLen)This function absorbs message data into a cSHAKE256 operation.
int	**wc_Cshake256_Final**(wc_Cshake * cshake, byte * out, word32 outLen)This function finalizes a cSHAKE256 operation, squeezing outLen bytes of output. cSHAKE is an XOF, so a longer squeeze extends a shorter one over the same input.
void	**wc_Cshake256_Free**(wc_Cshake * cshake)This function releases any resources associated with a cSHAKE256 operation. Passing NULL is safe and does nothing.
int	**wc_Cshake256**(const byte * name, word32 nameLen, const byte * custom, word32 customLen, const byte * in, word32 inLen, byte * out, word32 outLen)This function computes a cSHAKE256 over a single message in one call, initializing, updating and finalizing internally.
int	**wc_Cshake128_Copy**(wc_Cshake * src, wc_Cshake * dst)This function copies the state of a cSHAKE128 operation so it can be finalized more than once, for example over a common message prefix. dst must already be an initialized wc_Cshake.
int	**wc_Cshake256_Copy**(wc_Cshake * src, wc_Cshake * dst)This function copies the state of a cSHAKE256 operation so it can be finalized more than once, for example over a common message prefix. dst must already be an initialized wc_Cshake.

C.52.2 Functions Documentation

C.52.2.1 function wc_InitSha3_224

```
int wc_InitSha3_224(
    wc_Sha3 * sha3,
    void * heap,
    int devId
)
```

This function initializes SHA3-224. This is automatically called by wc_Sha3_224Hash.

Parameters:

- **sha3** pointer to the sha3 structure to use for encryption

See:

- wc_Sha3_224Hash
- wc_Sha3_224_Update
- wc_Sha3_224_Final

Return: 0 Returned upon successfully initializing

Example

```
wc_Sha3 sha3[1];
if ((ret = wc_InitSha3_224(sha3, NULL, INVALID_DEVID)) != 0) {
    WOLFSSL_MSG("wc_InitSha3_224 failed");
}
else {
    wc_Sha3_224_Update(sha3, data, len);
    wc_Sha3_224_Final(sha3, hash);
}
```

C.52.2.2 function wc_Sha3_224_Update

```
int wc_Sha3_224_Update(
    wc_Sha3 * sha,
    const byte * data,
    word32 len
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **sha3** pointer to the sha3 structure to use for encryption
- **data** the data to be hashed
- **len** length of data to be hashed

See:

- wc_Sha3_224Hash
- wc_Sha3_224_Final
- wc_InitSha3_224

Return: 0 Returned upon successfully adding the data to the digest.

Example

```
wc_Sha3 sha3[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha3_224(sha3, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitSha3_224 failed");
}
else {
    wc_Sha3_224_Update(sha3, data, len);
    wc_Sha3_224_Final(sha3, hash);
}
```

C.52.2.3 function wc_Sha3_224_Final

```
int wc_Sha3_224_Final(
    wc_Sha3 * sha3,
    byte * hash
)
```

Finalizes hashing of data. Result is placed into hash. Resets state of sha3 struct.

Parameters:

- **sha3** pointer to the sha3 structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Sha3_224Hash
- wc_Sha3_224_GetHash
- wc_InitSha3_224

Return: 0 Returned upon successfully finalizing.

Example

```
wc_Sha3 sha3[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha3_224(sha3, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitSha3_224 failed");
}
else {
   wc_Sha3_224_Update(sha3, data, len);
   wc_Sha3_224_Final(sha3, hash);
}
```

C.52.2.4 function wc_Sha3_224_Free

```
void wc_Sha3_224_Free(
    wc_Sha3 * sha3
)
```

Resets the wc_Sha3 structure. Note: this is only supported if you have WOLFSSL_TI_HASH defined.

Parameters:

- **sha3** Pointer to the sha3 structure to be freed.

See:

- wc_InitSha3_224
- wc_Sha3_224_Update
- wc_Sha3_224_Final

Return: none No returns.

Example

```
wc_Sha3 sha3;
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha3_224(&sha3, NULL, INVALID_DEVID)) != 0) {
```

```
    WOLFSSL_MSG("wc_InitSha3_224 failed");
}
else {
    wc_Sha3_224_Update(&sha3, data, len);
    wc_Sha3_224_Final(&sha3, hash);
    wc_Sha3_224_Free(&sha3);
}
```

C.52.2.5 function wc_Sha3_224_Reset

```
int wc_Sha3_224_Reset(
    wc_Sha3 * sha3
)
```

Resets the sha3 structure to its freshly initialized state, continuing with the established heap hint and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha3_224_Free for teardown at end of life.

Parameters:

- **sha3** Pointer to the sha3 structure to be reset.

See:

- wc_InitSha3_224
- wc_Sha3_224_Final
- wc_Sha3_224_Free

Return:

- 0 Returned upon successfully resetting the structure.
- BAD_FUNC_ARG Returned if sha3 is NULL.
- negative Other values may be returned on reinitialization failure in fallback (non-plain-software) configurations.

Example

```
wc_Sha3 sha3;
byte data[] = { Data to be discarded };
byte data2[] = { Data to be hashed };
byte hash[WC_SHA3_224_DIGEST_SIZE];

if ((ret = wc_InitSha3_224(&sha3, NULL, INVALID_DEVID)) == 0) {
    wc_Sha3_224_Update(&sha3, data, sizeof(data));
    wc_Sha3_224_Reset(&sha3); /* discard the partially hashed message */
    wc_Sha3_224_Update(&sha3, data2, sizeof(data2));
    wc_Sha3_224_Final(&sha3, hash);
    wc_Sha3_224_Free(&sha3);
}
```

C.52.2.6 function wc_Sha3_224_GetHash

```
int wc_Sha3_224_GetHash(
    wc_Sha3 * sha3,
    byte * hash
)
```

Gets hash data. Result is placed into hash. Does not reset state of sha3 struct.

Parameters:

- **sha3** pointer to the sha3 structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Sha3_224Hash
- wc_Sha3_224_Final
- wc_InitSha3_224
- wc_Sha3_224_Copy

Return: 0 Returned upon successful copying of the hash.

Example

```
wc_Sha3 sha3[1];
if ((ret = wc_InitSha3_224(sha3, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitSha3_224 failed");
}
else {
   wc_Sha3_224_Update(sha3, data, len);
   wc_Sha3_224_GetHash(sha3, hash);
}
```

C.52.2.7 function wc_Sha3_224_Copy

```
int wc_Sha3_224_Copy(
    wc_Sha3 * sha3,
    wc_Sha3 * dst
)
```

Copy the state of the hash.

Parameters:

- **sha3** pointer to the sha3 structure to copy
- **dst** pointer to the sha3 structure to copy into (must be zeroed or previously initialized)

See:

- wc_Sha3_224Hash
- wc_Sha3_224_Final
- wc_InitSha3_224
- wc_Sha3_224_GetHash

Return: 0 Returned upon successful copying.

Example

```
wc_Sha3 sha3[1];
wc_Sha3 sha3_dup[1];
memset(sha3_dup, 0, sizeof(*sha3_dup));
if ((ret = wc_InitSha3_224(sha3, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitSha3_224 failed");
}
else {
   wc_Sha3_224_Update(sha3, data, len);
   wc_Sha3_224_Copy(sha3, sha3_dup);
}
```

C.52.2.8 function wc_InitSha3_256

```
int wc_InitSha3_256(
    wc_Sha3 * sha3,
    void * heap,
    int devId
)
```

This function initializes SHA3-256. This is automatically called by wc_Sha3_256Hash.

Parameters:

- **sha3** pointer to the sha3 structure to use for encryption

See:

- wc_Sha3_256Hash
- wc_Sha3_256_Update
- wc_Sha3_256_Final

Return: 0 Returned upon successfully initializing

Example

```
wc_Sha3 sha3[1];
if ((ret = wc_InitSha3_256(sha3, NULL, INVALID_DEVID)) != 0) {
    WOLFSSL_MSG("wc_InitSha3_256 failed");
}
else {
    wc_Sha3_256_Update(sha3, data, len);
    wc_Sha3_256_Final(sha3, hash);
}
```

C.52.2.9 function wc_Sha3_256_Update

```
int wc_Sha3_256_Update(
    wc_Sha3 * sha,
    const byte * data,
    word32 len
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **sha3** pointer to the sha3 structure to use for encryption
- **data** the data to be hashed
- **len** length of data to be hashed

See:

- wc_Sha3_256Hash
- wc_Sha3_256_Final
- wc_InitSha3_256

Return: 0 Returned upon successfully adding the data to the digest.

Example

```
wc_Sha3 sha3[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);
```

```
if ((ret = wc_InitSha3_256(sha3, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitSha3_256 failed");
}
else {
    wc_Sha3_256_Update(sha3, data, len);
    wc_Sha3_256_Final(sha3, hash);
}
```

C.52.2.10 function wc_Sha3_256_Final

```
int wc_Sha3_256_Final(
    wc_Sha3 * sha3,
    byte * hash
)
```

Finalizes hashing of data. Result is placed into hash. Resets state of sha3 struct.

Parameters:

- **sha3** pointer to the sha3 structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Sha3_256Hash
- wc_Sha3_256_GetHash
- wc_InitSha3_256

Return: 0 Returned upon successfully finalizing.

Example

```
wc_Sha3 sha3[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha3_256(sha3, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitSha3_256 failed");
}
else {
   wc_Sha3_256_Update(sha3, data, len);
   wc_Sha3_256_Final(sha3, hash);
}
```

C.52.2.11 function wc_Sha3_256_Free

```
void wc_Sha3_256_Free(
    wc_Sha3 * sha3
)
```

Resets the wc_Sha3 structure. Note: this is only supported if you have WOLFSSL_TI_HASH defined.

Parameters:

- **sha3** Pointer to the sha3 structure to be freed.

See:

- wc_InitSha3_256
- wc_Sha3_256_Update
- wc_Sha3_256_Final

Return: none No returns.

Example

```
wc_Sha3 sha3;
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha3_256(&sha3, NULL, INVALID_DEVID)) != 0) {
    WOLFSSL_MSG("wc_InitSha3_256 failed");
}
else {
    wc_Sha3_256_Update(&sha3, data, len);
    wc_Sha3_256_Final(&sha3, hash);
    wc_Sha3_256_Free(&sha3);
}
```

C.52.2.12 function wc_Sha3_256_Reset

```
int wc_Sha3_256_Reset(
    wc_Sha3 * sha3
)
```

Resets the sha3 structure to its freshly initialized state, continuing with the established heap hint and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha3_256_Free for teardown at end of life.

Parameters:

- **sha3** Pointer to the sha3 structure to be reset.

See:

- wc_InitSha3_256
- wc_Sha3_256_Final
- wc_Sha3_256_Free

Return:

- 0 Returned upon successfully resetting the structure.
- BAD_FUNC_ARG Returned if sha3 is NULL.
- negative Other values may be returned on reinitialization failure in fallback (non-plain-software) configurations.

Example

```
wc_Sha3 sha3;
byte data[] = { Data to be discarded };
byte data2[] = { Data to be hashed };
byte hash[WC_SHA3_256_DIGEST_SIZE];

if ((ret = wc_InitSha3_256(&sha3, NULL, INVALID_DEVID)) == 0) {
    wc_Sha3_256_Update(&sha3, data, sizeof(data));
    wc_Sha3_256_Reset(&sha3); /* discard the partially hashed message */
    wc_Sha3_256_Update(&sha3, data2, sizeof(data2));
    wc_Sha3_256_Final(&sha3, hash);
    wc_Sha3_256_Free(&sha3);
}
```

C.52.2.13 function wc_Sha3_256_GetHash

```
int wc_Sha3_256_GetHash(
    wc_Sha3 * sha3,
    byte * hash
)
```

Gets hash data. Result is placed into hash. Does not reset state of sha3 struct.

Parameters:

- **sha3** pointer to the sha3 structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Sha3_256Hash
- wc_Sha3_256_Final
- wc_InitSha3_256
- wc_Sha3_256_Copy

Return: 0 Returned upon successful copying of the hash.

Example

```
wc_Sha3 sha3[1];
if ((ret = wc_InitSha3_256(sha3, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitSha3_256 failed");
}
else {
   wc_Sha3_256_Update(sha3, data, len);
   wc_Sha3_256_GetHash(sha3, hash);
}
```

C.52.2.14 function wc_Sha3_256_Copy

```
int wc_Sha3_256_Copy(
    wc_Sha3 * sha3,
    wc_Sha3 * dst
)
```

Copy the state of the hash.

Parameters:

- **sha3** pointer to the sha3 structure to copy
- **dst** pointer to the sha3 structure to copy into (must be zeroed or previously initialized)

See:

- wc_Sha3_256Hash
- wc_Sha3_256_Final
- wc_InitSha3_256
- wc_Sha3_256_GetHash

Return: 0 Returned upon successful copying.

Example

```
wc_Sha3 sha3[1];
wc_Sha3 sha3_dup[1];
memset(sha3_dup, 0, sizeof(*sha3_dup));
if ((ret = wc_InitSha3_256(sha3, NULL, INVALID_DEVID)) != 0) {
```

```
    WOLFSSL_MSG("wc_InitSha3_256 failed");
}
else {
    wc_Sha3_256_Update(sha3, data, len);
    wc_Sha3_256_Copy(sha3, sha3_dup);
}
```

C.52.2.15 function wc_InitSha3_384

```
int wc_InitSha3_384(
    wc_Sha3 * sha3,
    void * heap,
    int devId
)
```

This function initializes SHA3-384. This is automatically called by wc_Sha3_384Hash.

Parameters:

- **sha3** pointer to the sha3 structure to use for encryption

See:

- wc_Sha3_384Hash
- wc_Sha3_384_Update
- wc_Sha3_384_Final

Return: 0 Returned upon successfully initializing

Example

```
wc_Sha3 sha3[1];
if ((ret = wc_InitSha3_384(sha3, NULL, INVALID_DEVID)) != 0) {
    WOLFSSL_MSG("wc_InitSha3_384 failed");
}
else {
    wc_Sha3_384_Update(sha3, data, len);
    wc_Sha3_384_Final(sha3, hash);
}
```

C.52.2.16 function wc_Sha3_384_Update

```
int wc_Sha3_384_Update(
    wc_Sha3 * sha,
    const byte * data,
    word32 len
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **sha3** pointer to the sha3 structure to use for encryption
- **data** the data to be hashed
- **len** length of data to be hashed

See:

- wc_Sha3_384Hash
- wc_Sha3_384_Final
- wc_InitSha3_384

Return: 0 Returned upon successfully adding the data to the digest.

Example

```
wc_Sha3 sha3[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha3_384(sha3, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitSha3_384 failed");
}
else {
    wc_Sha3_384_Update(sha3, data, len);
    wc_Sha3_384_Final(sha3, hash);
}
```

C.52.2.17 function wc_Sha3_384_Final

```
int wc_Sha3_384_Final(
    wc_Sha3 * sha3,
    byte * hash
)
```

Finalizes hashing of data. Result is placed into hash. Resets state of sha3 struct.

Parameters:

- **sha3** pointer to the sha3 structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Sha3_384Hash
- wc_Sha3_384_GetHash
- wc_InitSha3_384

Return: 0 Returned upon successfully finalizing.

Example

```
wc_Sha3 sha3[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha3_384(sha3, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitSha3_384 failed");
}
else {
   wc_Sha3_384_Update(sha3, data, len);
   wc_Sha3_384_Final(sha3, hash);
}
```

C.52.2.18 function wc_Sha3_384_Free

```
void wc_Sha3_384_Free(
    wc_Sha3 * sha3
)
```

Resets the wc_Sha3 structure. Note: this is only supported if you have WOLFSSL_TI_HASH defined.

Parameters:

- **sha3** Pointer to the sha3 structure to be freed.

See:

- wc_InitSha3_384
- wc_Sha3_384_Update
- wc_Sha3_384_Final

Return: none No returns.

Example

```
wc_Sha3 sha3;
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha3_384(&sha3, NULL, INVALID_DEVID)) != 0) {
    WOLFSSL_MSG("wc_InitSha3_384 failed");
}
else {
    wc_Sha3_384_Update(&sha3, data, len);
    wc_Sha3_384_Final(&sha3, hash);
    wc_Sha3_384_Free(&sha3);
}
```

C.52.2.19 function wc_Sha3_384_Reset

```
int wc_Sha3_384_Reset(
    wc_Sha3 * sha3
)
```

Resets the sha3 structure to its freshly initialized state, continuing with the established heap hint and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha3_384_Free for teardown at end of life.

Parameters:

- **sha3** Pointer to the sha3 structure to be reset.

See:

- wc_InitSha3_384
- wc_Sha3_384_Final
- wc_Sha3_384_Free

Return:

- 0 Returned upon successfully resetting the structure.
- BAD_FUNC_ARG Returned if sha3 is NULL.
- negative Other values may be returned on reinitialization failure in fallback (non-plain-software) configurations.

Example

```
wc_Sha3 sha3;
byte data[] = { Data to be discarded };
byte data2[] = { Data to be hashed };
byte hash[WC_SHA3_384_DIGEST_SIZE];
```

```
if ((ret = wc_InitSha3_384(&sha3, NULL, INVALID_DEVID)) == 0) {
    wc_Sha3_384_Update(&sha3, data, sizeof(data));
    wc_Sha3_384_Reset(&sha3); /* discard the partially hashed message */
    wc_Sha3_384_Update(&sha3, data2, sizeof(data2));
    wc_Sha3_384_Final(&sha3, hash);
    wc_Sha3_384_Free(&sha3);
}
```

C.52.2.20 function wc_Sha3_384_GetHash

```
int wc_Sha3_384_GetHash(
    wc_Sha3 * sha3,
    byte * hash
)
```

Gets hash data. Result is placed into hash. Does not reset state of sha3 struct.

Parameters:

- **sha3** pointer to the sha3 structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Sha3_384Hash
- wc_Sha3_384_Final
- wc_InitSha3_384
- wc_Sha3_384_Copy

Return: 0 Returned upon successful copying of the hash.

Example

```
wc_Sha3 sha3[1];
if ((ret = wc_InitSha3_384(sha3, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitSha3_38384ailed");
}
else {
   wc_Sha3_384_Update(sha3, data, len);
   wc_Sha3_384_GetHash(sha3, hash);
}
```

C.52.2.21 function wc_Sha3_384_Copy

```
int wc_Sha3_384_Copy(
    wc_Sha3 * sha3,
    wc_Sha3 * dst
)
```

Copy the state of the hash.

Parameters:

- **sha3** pointer to the sha3 structure to copy
- **dst** pointer to the sha3 structure to copy into (must be zeroed or previously initialized)

See:

- wc_Sha3_384Hash
- wc_Sha3_384_Final
- wc_InitSha3_384

- wc_Sha3_384_GetHash

Return: 0 Returned upon successful copying.

Example

```
wc_Sha3 sha3[1];
wc_Sha3 sha3_dup[1];
memset(sha3_dup, 0, sizeof(*sha3_dup));
if ((ret = wc_InitSha3_384(sha3, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitSha3_384 failed");
}
else {
   wc_Sha3_384_Update(sha3, data, len);
   wc_Sha3_384_Copy(sha3, sha3_dup);
}
```

C.52.2.22 function wc_InitSha3_512

```
int wc_InitSha3_512(
    wc_Sha3 * sha3,
    void * heap,
    int devId
)
```

This function initializes SHA3-512. This is automatically called by wc_Sha3_512Hash.

Parameters:

- **sha3** pointer to the sha3 structure to use for encryption

See:

- wc_Sha3_512Hash
- wc_Sha3_512_Update
- wc_Sha3_512_Final

Return: 0 Returned upon successfully initializing

Example

```
wc_Sha3 sha3[1];
if ((ret = wc_InitSha3_512(sha3, NULL, INVALID_DEVID)) != 0) {
    WOLFSSL_MSG("wc_InitSha3_512 failed");
}
else {
    wc_Sha3_512_Update(sha3, data, len);
    wc_Sha3_512_Final(sha3, hash);
}
```

C.52.2.23 function wc_Sha3_512_Update

```
int wc_Sha3_512_Update(
    wc_Sha3 * sha,
    const byte * data,
    word32 len
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **sha3** pointer to the sha3 structure to use for encryption
- **data** the data to be hashed
- **len** length of data to be hashed

See:

- wc_Sha3_512Hash
- wc_Sha3_512_Final
- wc_InitSha3_512

Return: 0 Returned upon successfully adding the data to the digest.

Example

```
wc_Sha3 sha3[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha3_512(sha3, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitSha3_512 failed");
}
else {
    wc_Sha3_512_Update(sha3, data, len);
    wc_Sha3_512_Final(sha3, hash);
}
```

C.52.2.24 function wc_Sha3_512_Final

```
int wc_Sha3_512_Final(
    wc_Sha3 * sha3,
    byte * hash
)
```

Finalizes hashing of data. Result is placed into hash. Resets state of sha3 struct.

Parameters:

- **sha3** pointer to the sha3 structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Sha3_512Hash
- wc_Sha3_512_GetHash
- wc_InitSha3_512

Return: 0 Returned upon successfully finalizing.

Example

```
wc_Sha3 sha3[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha3_512(sha3, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitSha3_512 failed");
}
else {
   wc_Sha3_512_Update(sha3, data, len);
   wc_Sha3_512_Final(sha3, hash);
}
```

C.52.2.25 function wc_Sha3_512_Free

```
void wc_Sha3_512_Free(
    wc_Sha3 * sha3
)
```

Resets the wc_Sha3 structure. Note: this is only supported if you have WOLFSSL_TI_HASH defined.

Parameters:

- **sha3** Pointer to the sha3 structure to be freed.

See:

- wc_InitSha3_512
- wc_Sha3_512_Update
- wc_Sha3_512_Final

Return: none No returns.

Example

```
wc_Sha3 sha3;
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha3_512(&sha3, NULL, INVALID_DEVID)) != 0) {
    WOLFSSL_MSG("wc_InitSha3_512 failed");
}
else {
    wc_Sha3_512_Update(&sha3, data, len);
    wc_Sha3_512_Final(&sha3, hash);
    wc_Sha3_512_Free(&sha3);
}
```

C.52.2.26 function wc_Sha3_512_Reset

```
int wc_Sha3_512_Reset(
    wc_Sha3 * sha3
)
```

Resets the sha3 structure to its freshly initialized state, continuing with the established heap hint and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha3_512_Free for teardown at end of life.

Parameters:

- **sha3** Pointer to the sha3 structure to be reset.

See:

- wc_InitSha3_512
- wc_Sha3_512_Final
- wc_Sha3_512_Free

Return:

- 0 Returned upon successfully resetting the structure.
- BAD_FUNC_ARG Returned if sha3 is NULL.
- negative Other values may be returned on reinitialization failure in fallback (non-plain-software) configurations.

Example

```
wc_Sha3 sha3;
byte data[] = { Data to be discarded };
byte data2[] = { Data to be hashed };
byte hash[WC_SHA3_512_DIGEST_SIZE];

if ((ret = wc_InitSha3_512(&sha3, NULL, INVALID_DEVID)) == 0) {
    wc_Sha3_512_Update(&sha3, data, sizeof(data));
    wc_Sha3_512_Reset(&sha3); /* discard the partially hashed message */
    wc_Sha3_512_Update(&sha3, data2, sizeof(data2));
    wc_Sha3_512_Final(&sha3, hash);
    wc_Sha3_512_Free(&sha3);
}
```

C.52.2.27 function wc_Sha3_512_GetHash

```
int wc_Sha3_512_GetHash(
    wc_Sha3 * sha3,
    byte * hash
)
```

Gets hash data. Result is placed into hash. Does not reset state of sha3 struct.

Parameters:

- **sha3** pointer to the sha3 structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Sha3_512Hash
- wc_Sha3_512_Final
- wc_InitSha3_512
- wc_Sha3_512_Copy

Return: 0 Returned upon successful copying of the hash.

Example

```
wc_Sha3 sha3[1];
if ((ret = wc_InitSha3_512(sha3, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitSha3_512 failed");
}
else {
   wc_Sha3_512_Update(sha3, data, len);
   wc_Sha3_512_GetHash(sha3, hash);
}
```

C.52.2.28 function wc_Sha3_512_Copy

```
int wc_Sha3_512_Copy(
    wc_Sha3 * sha3,
    wc_Sha3 * dst
)
```

Copy the state of the hash.

Parameters:

- **sha3** pointer to the sha3 structure to copy
- **dst** pointer to the sha3 structure to copy into (must be zeroed or previously initialized)

See:

- wc_Sha3_512Hash
- wc_Sha3_512_Final
- wc_InitSha3_512
- wc_Sha3_512_GetHash

Return: 0 Returned upon successful copying.

Example

```
wc_Sha3 sha3[1];
wc_Sha3 sha3_dup[1];
memset(sha3_dup, 0, sizeof(*sha3_dup));
if ((ret = wc_InitSha3_512(sha3, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitSha3_512 failed");
}
else {
   wc_Sha3_512_Update(sha3, data, len);
   wc_Sha3_512_Copy(sha3, sha3_dup);
}
```

C.52.2.29 function wc_InitShake128

```
int wc_InitShake128(
    wc_Shake * shake,
    void * heap,
    int devId
)
```

This function initializes SHAKE-128. This is automatically called by wc_Shake128Hash.

Parameters:

- **shake** pointer to the shake structure to use for encryption

See:

- wc_Shake128Hash
- wc_Shake128_Update
- wc_Shake128_Final

Return: 0 Returned upon successfully initializing

Example

```
wc_Shake shake[1];
if ((ret = wc_InitShake128(shake, NULL, INVALID_DEVID)) != 0) {
    WOLFSSL_MSG("wc_InitShake128 failed");
}
else {
    wc_Shake128_Update(shake, data, len);
    wc_Shake128_Final(shake, hash);
}
```

C.52.2.30 function wc_Shake128_Update

```
int wc_Shake128_Update(
    wc_Shake * sha,
    const byte * data,
    word32 len
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **shake** pointer to the shake structure to use for encryption
- **data** the data to be hashed
- **len** length of data to be hashed

See:

- wc_Shake128Hash
- wc_Shake128_Final
- wc_InitShake128

Return: 0 Returned upon successfully adding the data to the digest.

Example

```
wc_Shake shake[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitShake128(shake, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitShake128 failed");
}
else {
    wc_Shake128_Update(shake, data, len);
    wc_Shake128_Final(shake, hash);
}
```

C.52.2.31 function wc_Shake128_Final

```
int wc_Shake128_Final(
    wc_Shake * shake,
    byte * hash,
    word32 hashLen
)
```

Finalizes hashing of data. Result is placed into hash. Resets state of shake struct.

Parameters:

- **shake** pointer to the shake structure to use for encryption
- **hash** Byte array to hold hash value.
- **hashLen** Number of bytes to write to hash.

See:

- wc_Shake128Hash
- wc_Shake128_GetHash
- wc_InitShake128

Return: 0 Returned upon successfully finalizing.

Example

```
wc_Shake shake[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitShake128(shake, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitShake128 failed");
}
else {
   wc_Shake128_Update(shake, data, len);
   wc_Shake128_Final(shake, hash);
}
```

C.52.2.32 function wc_Shake128_Absorb

```
int wc_Shake128_Absorb(
    wc_Shake * sha,
    const byte * data,
    word32 len
)
```

Called to absorb the provided byte array of length len. Cannot be called incrementally.

Parameters:

- **shake** pointer to the shake structure to use for encryption
- **data** the data to be absorbed
- **len** length of data to be absorbed

See:

- wc_Shake128_SqueezeBlocks
- wc_InitShake128

Return: 0 Returned upon successfully absorbed the data.

Example

```
wc_Shake shake[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);
byte out[2 * WC_SHA3_128_BLOCK_SIZE];
int blocks = 2;

if ((ret = wc_InitShake128(shake, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitShake128 failed");
}
else {
   wc_Shake128_Absorb(shake, data, len);
   wc_Shake128_SqueezeBlocks(shake, out, blocks);
}
```

C.52.2.33 function wc_Shake128_SqueezeBlocks

```
int wc_Shake128_SqueezeBlocks(
    wc_Shake * shake,
    byte * out,
```

```
    word32 blockCnt
)
```

Squeeze out more blocks of data. Result is placed into out. Can be called inrementally.

Parameters:

- **shake** pointer to the shake structure to use for encryption
- **hash** Byte array to hold output.
- **blocks** Number of blocks to squeeze out. Each block is WC_SHA3_128_BLOCK_SIZE bytes in length.

See:

- wc_Shake128_Absorb
- wc_InitShake128

Return: 0 Returned upon successfully squeezing.

Example

```
wc_Shake shake[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);
byte out[2 * WC_SHA3_128_BLOCK_SIZE];
int blocks = 2;

if ((ret = wc_InitShake128(shake, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitShake128 failed");
}
else {
   wc_Shake128_Absorb(shake, data, len);
   wc_Shake128_SqueezeBlocks(shake, out, blocks);
}
```

C.52.2.34 function wc_Shake128_Free

```
void wc_Shake128_Free(
    wc_Shake * shake
)
```

Resets the wc_Shake structure. Note: this is only supported if you have WOLFSSL_TI_HASH defined.

Parameters:

- **shake** Pointer to the shake structure to be freed.

See:

- wc_InitShake128
- wc_Shake128_Update
- wc_Shake128_Final

Return: none No returns.

Example

```
wc_Shake shake;
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitShake128(&shake, NULL, INVALID_DEVID)) != 0) {
```

```
    WOLFSSL_MSG("wc_InitShake128 failed");
}
else {
    wc_Shake128_Update(&shake, data, len);
    wc_Shake128_Final(&shake, hash);
    wc_Shake128_Free(&shake);
}
```

C.52.2.35 function wc_Shake128_Reset

```
int wc_Shake128_Reset(
    wc_Shake * shake
)
```

Resets the shake structure to its freshly initialized state, continuing with the established heap hint and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially absorbed input, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Shake128_Free for teardown at end of life.

Parameters:

- **shake** Pointer to the shake structure to be reset.

See:

- wc_InitShake128
- wc_Shake128_Final
- wc_Shake128_Free

Return:

- 0 Returned upon successfully resetting the structure.
- BAD_FUNC_ARG Returned if shake is NULL.
- negative Other values may be returned on reinitialization failure in fallback (non-plain-software) configurations.

Example

```
wc_Shake shake;
byte data[] = { Data to be discarded };
byte data2[] = { Data to be hashed };
byte hash[64];

if ((ret = wc_InitShake128(&shake, NULL, INVALID_DEVID)) == 0) {
    wc_Shake128_Update(&shake, data, sizeof(data));
    wc_Shake128_Reset(&shake); /* discard the partially absorbed input */
    wc_Shake128_Update(&shake, data2, sizeof(data2));
    wc_Shake128_Final(&shake, hash, sizeof(hash));
    wc_Shake128_Free(&shake);
}
```

C.52.2.36 function wc_Shake128_GetHash

```
int wc_Shake128_GetHash(
    wc_Shake * shake,
    byte * hash
)
```

Gets hash data. Result is placed into hash. Does not reset state of shake struct.

Parameters:

- **shake** pointer to the shake structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Shake128Hash
- wc_Shake128_Final
- wc_InitShake128
- wc_Shake128_Copy

Return: 0 Returned upon successful copying of the hash.

Example

```
wc_Shake shake[1];
if ((ret = wc_InitShake128(shake, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitShake128 failed");
}
else {
   wc_Shake128_Update(shake, data, len);
   wc_Shake128_GetHash(shake, hash);
}
```

C.52.2.37 function wc_Shake128_Copy

```
int wc_Shake128_Copy(
    wc_Shake * src,
    wc_Sha3 * dst
)
```

Copy the state of the hash.

Parameters:

- **shake** pointer to the shake structure to copy
- **dst** pointer to the shake structure to copy into (must be zeroed or previously initialized)

See:

- wc_Shake128Hash
- wc_Shake128_Final
- wc_InitShake128
- wc_Shake128_GetHash

Return: 0 Returned upon successful copying.

Example

```
wc_Shake shake[1];
wc_Shake shake_dup[1];
memset(shake_dup, 0, sizeof(*shake_dup));
if ((ret = wc_InitShake128(shake, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitShake128 failed");
}
else {
   wc_Shake128_Update(shake, data, len);
   wc_Shake128_Copy(shake, shake_dup);
}
```

C.52.2.38 function wc_InitShake256

```
int wc_InitShake256(
    wc_Shake * shake,
    void * heap,
    int devId
)
```

This function initializes SHAKE-256. This is automatically called by wc_Shake256Hash.

Parameters:

- **shake** pointer to the shake structure to use for encryption

See:

- wc_Shake256Hash
- wc_Shake256_Update
- wc_Shake256_Final

Return: 0 Returned upon successfully initializing

Example

```
wc_Shake shake[1];
if ((ret = wc_InitShake256(shake, NULL, INVALID_DEVID)) != 0) {
    WOLFSSL_MSG("wc_InitShake256 failed");
}
else {
    wc_Shake256_Update(shake, data, len);
    wc_Shake256_Final(shake, hash, sizeof(hash));
}
```

C.52.2.39 function wc_Shake256_Update

```
int wc_Shake256_Update(
    wc_Shake * sha,
    const byte * data,
    word32 len
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **shake** pointer to the shake structure to use for encryption
- **data** the data to be hashed
- **len** length of data to be hashed

See:

- wc_Shake256Hash
- wc_Shake256_Final
- wc_InitShake256

Return: 0 Returned upon successfully adding the data to the digest.

Example

```
wc_Shake shake[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);
```

```
if ((ret = wc_InitShake256(shake, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitShake256 failed");
}
else {
    wc_Shake256_Update(shake, data, len);
    wc_Shake256_Final(shake, hash, sizeof(hash));
}
```

C.52.2.40 function wc_Shake256_Final

```
int wc_Shake256_Final(
    wc_Shake * shake,
    byte * hash,
    word32 hashLen
)
```

Finalizes hashing of data. Result is placed into hash. Resets state of shake struct.

Parameters:

- **shake** pointer to the shake structure to use for encryption
- **hash** Byte array to hold hash value.
- **hashLen** Size of hash in bytes.

See:

- wc_Shake256Hash
- wc_Shake256_GetHash
- wc_InitShake256

Return: 0 Returned upon successfully finalizing.

Example

```
wc_Shake shake[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitShake256(shake, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitShake256 failed");
}
else {
   wc_Shake256_Update(shake, data, len);
   wc_Shake256_Final(shake, hash, sizeof(hash));
}
```

C.52.2.41 function wc_Shake256_Absorb

```
int wc_Shake256_Absorb(
    wc_Shake * sha,
    const byte * data,
    word32 len
)
```

Called to absorb the provided byte array of length len. Cannot be called incrementally.

Parameters:

- **shake** pointer to the shake structure to use for encryption
- **data** the data to be absorbed

- **len** length of data to be absorbed

See:

- wc_Shake256_SqueezeBlocks
- wc_InitShake256

Return: 0 Returned upon successfully absorbed the data.

Example

```
wc_Shake shake[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);
byte out[2 * WC_SHA3_256_BLOCK_SIZE];
int blocks = 2;

if ((ret = wc_InitShake256(shake, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitShake256 failed");
}
else {
   wc_Shake256_Absorb(shake, data, len);
   wc_Shake256_SqueezeBlocks(shake, out, blocks);
}
```

C.52.2.42 function wc_Shake256_SqueezeBlocks

```
int wc_Shake256_SqueezeBlocks(
    wc_Shake * shake,
    byte * out,
    word32 blockCnt
)
```

Squeeze out more blocks of data. Result is placed into out. Can be called incrementally.

Parameters:

- **shake** pointer to the shake structure to use for encryption
- **hash** Byte array to hold output.
- **blocks** Number of blocks to squeeze out. Each block is WC_SHA3_256_BLOCK_SIZE bytes in length.

See:

- wc_Shake256_Absorb
- wc_InitShake256

Return: 0 Returned upon successfully squeezing.

Example

```
wc_Shake shake[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);
byte out[2 * WC_SHA3_256_BLOCK_SIZE];
int blocks = 2;

if ((ret = wc_InitShake256(shake, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitShake256 failed");
}
else {
```

```
    wc_Shake256_Absorb(shake, data, len);
    wc_Shake256_SqueezeBlocks(shake, out, blocks);
}
```

C.52.2.43 function wc_Shake256_Free

```
void wc_Shake256_Free(
    wc_Shake * shake
)
```

Resets the wc_Shake structure. Note: this is only supported if you have WOLFSSL_TI_HASH defined.

Parameters:

- **shake** Pointer to the shake structure to be freed.

See:

- wc_InitShake256
- wc_Shake256_Update
- wc_Shake256_Final

Return: none No returns.

Example

```
wc_Shake shake;
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitShake256(&shake, NULL, INVALID_DEVID)) != 0) {
    WOLFSSL_MSG("wc_InitShake256 failed");
}
else {
    wc_Shake256_Update(&shake, data, len);
    wc_Shake256_Final(&shake, hash, sizeof(hash));
    wc_Shake256_Free(&shake);
}
```

C.52.2.44 function wc_Shake256_Reset

```
int wc_Shake256_Reset(
    wc_Shake * shake
)
```

Resets the shake structure to its freshly initialized state, continuing with the established heap hint and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially absorbed input, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Shake256_Free for teardown at end of life.

Parameters:

- **shake** Pointer to the shake structure to be reset.

See:

- wc_InitShake256
- wc_Shake256_Final
- wc_Shake256_Free

Return:

- 0 Returned upon successfully resetting the structure.
- BAD_FUNC_ARG Returned if shake is NULL.
- negative Other values may be returned on reinitialization failure in fallback (non-plain-software) configurations.

Example

```
wc_Shake shake;
byte data[] = { Data to be discarded };
byte data2[] = { Data to be hashed };
byte hash[64];

if ((ret = wc_InitShake256(&shake, NULL, INVALID_DEVID)) == 0) {
    wc_Shake256_Update(&shake, data, sizeof(data));
    wc_Shake256_Reset(&shake); /* discard the partially absorbed input */
    wc_Shake256_Update(&shake, data2, sizeof(data2));
    wc_Shake256_Final(&shake, hash, sizeof(hash));
    wc_Shake256_Free(&shake);
}
```

C.52.2.45 function wc_Shake256_GetHash

```
int wc_Shake256_GetHash(
    wc_Shake * shake,
    byte * hash
)
```

Gets hash data. Result is placed into hash. Does not reset state of shake struct.

Parameters:

- **shake** pointer to the shake structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Shake256Hash
- wc_Shake256_Final
- wc_InitShake256
- wc_Shake256_Copy

Return: 0 Returned upon successful copying of the hash.

Example

```
wc_Shake shake[1];
if ((ret = wc_InitShake256(shake, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitShake256 failed");
}
else {
   wc_Shake256_Update(shake, data, len);
   wc_Shake256_GetHash(shake, hash);
}
```

C.52.2.46 function wc_Shake256_Copy

```
int wc_Shake256_Copy(
    wc_Shake * src,
```

```
    wc_Sha3 * dst
)
```

Copy the state of the hash.

Parameters:

- **shake** pointer to the shake structure to copy
- **dst** pointer to the shake structure to copy into (must be zeroed or previously initialized)

See:

- wc_Shake256Hash
- wc_Shake256_Final
- wc_InitShake256
- wc_Shake256_GetHash

Return: 0 Returned upon successful copying.

Example

```
wc_Shake shake[1];
wc_Shake shake_dup[1];
memset(shake_dup, 0, sizeof(*shake_dup));
if ((ret = wc_InitShake256(shake, NULL, INVALID_DEVID)) != 0) {
   WOLFSSL_MSG("wc_InitShake256 failed");
}
else {
   wc_Shake256_Update(shake, data, len);
   wc_Shake256_Copy(shake, shake_dup);
}
```

C.52.2.47 function wc_InitKmac128

```
int wc_InitKmac128(
    wc_Kmac * kmac,
    const byte * key,
    word32 keyLen,
    const byte * custom,
    word32 customLen,
    void * heap,
    int devId
)
```

This function initializes a KMAC128 operation with a key and an optional customization string, per NIST SP 800-185. KMAC128 requires SHAKE128 (WOLFSSL_SHAKE128) and is enabled with WOLFSSL_KMAC.

Parameters:

- **kmac** pointer to the wc_Kmac structure to initialize.
- **key** the key bytes.
- **keyLen** length of the key in bytes.
- **custom** customization string, or NULL when customLen is 0.
- **customLen** length of the customization string in bytes.
- **heap** pointer to a heap hint, may be NULL.
- **devId** device identifier, or INVALID_DEVID.

See:

- wc_Kmac128_Update
- wc_Kmac128_Final

- wc_Kmac128_FinalXof
- wc_Kmac128_Free
- wc_Kmac128Hash

Return:

- 0 Returned upon successfully initializing.
- BAD_FUNC_ARG Returned when kmac is NULL, or key/custom is NULL with a non-zero length.

Example

```
wc_Kmac kmac;
byte key[32];
byte data[] = { Data to authenticate };
byte out[32];

if (wc_InitKmac128(&kmac, key, sizeof(key), NULL, 0, NULL,
        INVALID_DEVID) == 0) {
    wc_Kmac128_Update(&kmac, data, sizeof(data));
    wc_Kmac128_Final(&kmac, out, sizeof(out));
    wc_Kmac128_Free(&kmac);
}
```

C.52.2.48 function wc_Kmac128_Update

```
int wc_Kmac128_Update(
    wc_Kmac * kmac,
    const byte * in,
    word32 inLen
)
```

This function absorbs message data into a KMAC128 operation. It may be called any number of times before finalizing.

Parameters:

- **kmac** pointer to the wc_Kmac structure holding state.
- **in** the message data to authenticate.
- **inLen** length of the message data in bytes.

See:

- wc_InitKmac128
- wc_Kmac128_Final
- wc_Kmac128_Free

Return:

- 0 Returned upon successfully adding the data.
- BAD_FUNC_ARG Returned when kmac is NULL, or in is NULL with a non-zero length.

Example

```
wc_Kmac kmac;
byte data[] = { Data to authenticate };

if (wc_InitKmac128(&kmac, key, keyLen, NULL, 0, NULL,
        INVALID_DEVID) == 0) {
    wc_Kmac128_Update(&kmac, data, sizeof(data));
    wc_Kmac128_Final(&kmac, out, sizeof(out));
```

```
    wc_Kmac128_Free(&kmac);
}
```

C.52.2.49 function wc_Kmac128_Final

```
int wc_Kmac128_Final(
    wc_Kmac * kmac,
    byte * out,
    word32 outLen
)
```

This function finalizes a KMAC128 operation, producing outLen bytes of output. The output length is bound into the result, so a KMAC with a different length is a different value.

Parameters:

- **kmac** pointer to the wc_Kmac structure holding state.
- **out** buffer to hold the output.
- **outLen** number of output bytes to produce.

See:

- wc_InitKmac128
- wc_Kmac128_Update
- wc_Kmac128_FinalXof
- wc_Kmac128_Free

Return:

- 0 Returned upon successfully finalizing.
- BAD_FUNC_ARG Returned when kmac is NULL, or out is NULL with a non-zero length.

Example

```
byte out[32];
wc_Kmac128_Final(&kmac, out, sizeof(out));
```

C.52.2.50 function wc_Kmac128_FinalXof

```
int wc_Kmac128_FinalXof(
    wc_Kmac * kmac,
    byte * out,
    word32 outLen
)
```

This function finalizes a KMAC128 operation as an XOF (KMACXOF128). Unlike wc_Kmac128_Final, the output length is not bound into the result, so any amount of output may be requested.

Parameters:

- **kmac** pointer to the wc_Kmac structure holding state.
- **out** buffer to hold the output.
- **outLen** number of output bytes to produce.

See:

- wc_InitKmac128
- wc_Kmac128_Update
- wc_Kmac128_Final
- wc_Kmac128_Free

Return:

- 0 Returned upon successfully finalizing.
- BAD_FUNC_ARG Returned when kmac is NULL, or out is NULL with a non-zero length.

Example

```
byte out[64];
wc_Kmac128_FinalXof(&kmac, out, sizeof(out));
```

C.52.2.51 function wc_Kmac128_Free

```
void wc_Kmac128_Free(
    wc_Kmac * kmac
)
```

This function releases any resources associated with a KMAC128 operation. Passing NULL is safe and does nothing.

Parameters:

- **kmac** pointer to the wc_Kmac structure to free, may be NULL.

See: wc_InitKmac128

Example

```
wc_Kmac128_Free(&kmac);
```

C.52.2.52 function wc_Kmac128Hash

```
int wc_Kmac128Hash(
    const byte * key,
    word32 keyLen,
    const byte * custom,
    word32 customLen,
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen
)
```

This function computes a KMAC128 over a single message in one call, initializing, updating and finalizing internally.

Parameters:

- **key** the key bytes.
- **keyLen** length of the key in bytes.
- **custom** customization string, or NULL when customLen is 0.
- **customLen** length of the customization string in bytes.
- **in** the message data to authenticate.
- **inLen** length of the message data in bytes.
- **out** buffer to hold the output.
- **outLen** number of output bytes to produce.

See:

- wc_InitKmac128
- wc_Kmac128_Update
- wc_Kmac128_Final

Return:

- 0 Returned upon success.
- BAD_FUNC_ARG Returned when a required pointer is NULL with a non-zero length.

Example

```
byte key[32];
byte data[] = { Data to authenticate };
byte out[32];

wc_Kmac128Hash(key, sizeof(key), NULL, 0, data, sizeof(data),
    out, sizeof(out));
```

C.52.2.53 function wc_Kmac128HashXof

```
int wc_Kmac128HashXof(
    const byte * key,
    word32 keyLen,
    const byte * custom,
    word32 customLen,
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen
)
```

This function computes a KMACXOF128 over a single message in one call. Like wc_Kmac128Hash but the output length is not bound into the result, so any amount of output may be requested.

Parameters:

- **key** the key bytes.
- **keyLen** length of the key in bytes.
- **custom** customization string, or NULL when customLen is 0.
- **customLen** length of the customization string in bytes.
- **in** the message data to authenticate.
- **inLen** length of the message data in bytes.
- **out** buffer to hold the output.
- **outLen** number of output bytes to produce.

See:

- wc_InitKmac128
- wc_Kmac128_FinalXof
- wc_Kmac128Hash

Return:

- 0 Returned upon success.
- BAD_FUNC_ARG Returned when a required pointer is NULL with a non-zero length.

Example

```
byte key[32];
byte data[] = { Data to authenticate };
byte out[64];

wc_Kmac128HashXof(key, sizeof(key), NULL, 0, data, sizeof(data),
    out, sizeof(out));
```

C.52.2.54 function wc_InitKmac256

```
int wc_InitKmac256(
    wc_Kmac * kmac,
    const byte * key,
    word32 keyLen,
    const byte * custom,
    word32 customLen,
    void * heap,
    int devId
)
```

This function initializes a KMAC256 operation with a key and an optional customization string, per NIST SP 800-185. KMAC256 requires SHAKE256 (WOLFSSL_SHAKE256) and is enabled with WOLFSSL_KMAC.

Parameters:

- **kmac** pointer to the wc_Kmac structure to initialize.
- **key** the key bytes.
- **keyLen** length of the key in bytes.
- **custom** customization string, or NULL when customLen is 0.
- **customLen** length of the customization string in bytes.
- **heap** pointer to a heap hint, may be NULL.
- **devId** device identifier, or INVALID_DEVID.

See:

- wc_Kmac256_Update
- wc_Kmac256_Final
- wc_Kmac256_FinalXof
- wc_Kmac256_Free
- wc_Kmac256Hash

Return:

- 0 Returned upon successfully initializing.
- BAD_FUNC_ARG Returned when kmac is NULL, or key/custom is NULL with a non-zero length.

Example

```
wc_Kmac kmac;
byte key[32];
byte data[] = { Data to authenticate };
byte out[64];

if (wc_InitKmac256(&kmac, key, sizeof(key), NULL, 0, NULL,
        INVALID_DEVID) == 0) {
    wc_Kmac256_Update(&kmac, data, sizeof(data));
    wc_Kmac256_Final(&kmac, out, sizeof(out));
    wc_Kmac256_Free(&kmac);
}
```

C.52.2.55 function wc_Kmac256_Update

```
int wc_Kmac256_Update(
    wc_Kmac * kmac,
    const byte * in,
    word32 inLen
)
```

This function absorbs message data into a KMAC256 operation. It may be called any number of times before finalizing.

Parameters:

- **kmac** pointer to the wc_Kmac structure holding state.
- **in** the message data to authenticate.
- **inLen** length of the message data in bytes.

See:

- wc_InitKmac256
- wc_Kmac256_Final
- wc_Kmac256_Free

Return:

- 0 Returned upon successfully adding the data.
- BAD_FUNC_ARG Returned when kmac is NULL, or in is NULL with a non-zero length.

Example

```
wc_Kmac256_Update(&kmac, data, sizeof(data));
```

C.52.2.56 function wc_Kmac256_Final

```
int wc_Kmac256_Final(
    wc_Kmac * kmac,
    byte * out,
    word32 outLen
)
```

This function finalizes a KMAC256 operation, producing outLen bytes of output. The output length is bound into the result, so a KMAC with a different length is a different value.

Parameters:

- **kmac** pointer to the wc_Kmac structure holding state.
- **out** buffer to hold the output.
- **outLen** number of output bytes to produce.

See:

- wc_InitKmac256
- wc_Kmac256_Update
- wc_Kmac256_FinalXof
- wc_Kmac256_Free

Return:

- 0 Returned upon successfully finalizing.
- BAD_FUNC_ARG Returned when kmac is NULL, or out is NULL with a non-zero length.

Example

```
byte out[64];
wc_Kmac256_Final(&kmac, out, sizeof(out));
```

C.52.2.57 function wc_Kmac256_FinalXof

```
int wc_Kmac256_FinalXof(
    wc_Kmac * kmac,
    byte * out,
```

```
    word32 outLen
)
```

This function finalizes a KMAC256 operation as an XOF (KMACXOF256). Unlike wc_Kmac256_Final, the output length is not bound into the result, so any amount of output may be requested.

Parameters:

- **kmac** pointer to the wc_Kmac structure holding state.
- **out** buffer to hold the output.
- **outLen** number of output bytes to produce.

See:

- wc_InitKmac256
- wc_Kmac256_Update
- wc_Kmac256_Final
- wc_Kmac256_Free

Return:

- 0 Returned upon successfully finalizing.
- BAD_FUNC_ARG Returned when kmac is NULL, or out is NULL with a non-zero length.

Example

```
byte out[128];
wc_Kmac256_FinalXof(&kmac, out, sizeof(out));
```

C.52.2.58 function wc_Kmac256_Free

```
void wc_Kmac256_Free(
    wc_Kmac * kmac
)
```

This function releases any resources associated with a KMAC256 operation. Passing NULL is safe and does nothing.

Parameters:

- **kmac** pointer to the wc_Kmac structure to free, may be NULL.

See: wc_InitKmac256

Example

```
wc_Kmac256_Free(&kmac);
```

C.52.2.59 function wc_Kmac256Hash

```
int wc_Kmac256Hash(
    const byte * key,
    word32 keyLen,
    const byte * custom,
    word32 customLen,
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen
)
```

This function computes a KMAC256 over a single message in one call, initializing, updating and finalizing internally.

Parameters:

- **key** the key bytes.
- **keyLen** length of the key in bytes.
- **custom** customization string, or NULL when customLen is 0.
- **customLen** length of the customization string in bytes.
- **in** the message data to authenticate.
- **inLen** length of the message data in bytes.
- **out** buffer to hold the output.
- **outLen** number of output bytes to produce.

See:

- wc_InitKmac256
- wc_Kmac256_Update
- wc_Kmac256_Final

Return:

- 0 Returned upon success.
- BAD_FUNC_ARG Returned when a required pointer is NULL with a non-zero length.

Example

```
byte key[32];
byte data[] = { Data to authenticate };
byte out[64];

wc_Kmac256Hash(key, sizeof(key), NULL, 0, data, sizeof(data),
    out, sizeof(out));
```

C.52.2.60 function wc_Kmac256HashXof

```
int wc_Kmac256HashXof(
    const byte * key,
    word32 keyLen,
    const byte * custom,
    word32 customLen,
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen
)
```

This function computes a KMACXOF256 over a single message in one call. Like wc_Kmac256Hash but the output length is not bound into the result, so any amount of output may be requested.

Parameters:

- **key** the key bytes.
- **keyLen** length of the key in bytes.
- **custom** customization string, or NULL when customLen is 0.
- **customLen** length of the customization string in bytes.
- **in** the message data to authenticate.
- **inLen** length of the message data in bytes.
- **out** buffer to hold the output.
- **outLen** number of output bytes to produce.

See:

- wc_InitKmac256
- wc_Kmac256_FinalXof
- wc_Kmac256Hash

Return:

- 0 Returned upon success.
- BAD_FUNC_ARG Returned when a required pointer is NULL with a non-zero length.

Example

```
byte key[32];
byte data[] = { Data to authenticate };
byte out[128];

wc_Kmac256HashXof(key, sizeof(key), NULL, 0, data, sizeof(data),
    out, sizeof(out));
```

C.52.2.61 function wc_Kmac128_Copy

```
int wc_Kmac128_Copy(
    wc_Kmac * src,
    wc_Kmac * dst
)
```

This function copies the state of a KMAC128 operation so it can be finalized more than once, for example over a common message prefix.

Parameters:

- **src** pointer to the wc_Kmac structure to copy from.
- **dst** pointer to the wc_Kmac structure to copy into.

See:

- wc_InitKmac128
- wc_Kmac128_Update
- wc_Kmac128_Final

Return:

- 0 Returned upon success.
- BAD_FUNC_ARG Returned when src or dst is NULL.

Example

```
wc_Kmac kmac;
wc_Kmac copy;

wc_InitKmac128(&kmac, key, keyLen, NULL, 0, NULL, INVALID_DEVID);
wc_Kmac128_Update(&kmac, prefix, prefixLen);
wc_Kmac128_Copy(&kmac, &copy);
```

C.52.2.62 function wc_Kmac256_Copy

```
int wc_Kmac256_Copy(
    wc_Kmac * src,
    wc_Kmac * dst
)
```

This function copies the state of a KMAC256 operation so it can be finalized more than once, for example over a common message prefix.

Parameters:

- **src** pointer to the wc_Kmac structure to copy from.
- **dst** pointer to the wc_Kmac structure to copy into.

See:

- wc_InitKmac256
- wc_Kmac256_Update
- wc_Kmac256_Final

Return:

- 0 Returned upon success.
- BAD_FUNC_ARG Returned when src or dst is NULL.

Example

```
wc_Kmac kmac;
wc_Kmac copy;

wc_InitKmac256(&kmac, key, keyLen, NULL, 0, NULL, INVALID_DEVID);
wc_Kmac256_Update(&kmac, prefix, prefixLen);
wc_Kmac256_Copy(&kmac, &copy);
```

C.52.2.63 function wc_InitCshake128

```
int wc_InitCshake128(
    wc_Cshake * cshake,
    const byte * name,
    word32 nameLen,
    const byte * custom,
    word32 customLen,
    void * heap,
    int devId
)
```

This function initializes a cSHAKE128 operation (customizable SHAKE, NIST SP 800-185) with a function-name and customization string. cSHAKE is enabled together with KMAC (WOLFSSL_KMAC). When both strings are empty, cSHAKE reduces to plain SHAKE128.

Parameters:

- **cshake** pointer to the wc_Cshake structure to initialize.
- **name** function-name string (reserved for NIST-defined functions); use an empty string for application customization via custom.
- **nameLen** length of the function-name string in bytes.
- **custom** customization string, or NULL when customLen is 0.
- **customLen** length of the customization string in bytes.
- **heap** pointer to a heap hint, may be NULL.
- **devId** device identifier, or INVALID_DEVID.

See:

- wc_Cshake128_Update
- wc_Cshake128_Final
- wc_Cshake128_Free
- wc_Cshake128

Return:

- 0 Returned upon successfully initializing.
- BAD_FUNC_ARG Returned when a required pointer is NULL.

Example

```
wc_Cshake cshake;
byte out[32];

if (wc_InitCshake128(&cshake, NULL, 0, custom, customLen, NULL,
        INVALID_DEVID) == 0) {
    wc_Cshake128_Update(&cshake, data, dataLen);
    wc_Cshake128_Final(&cshake, out, sizeof(out));
    wc_Cshake128_Free(&cshake);
}
```

C.52.2.64 function wc_Cshake128_Update

```
int wc_Cshake128_Update(
    wc_Cshake * cshake,
    const byte * in,
    word32 inLen
)
```

This function absorbs message data into a cSHAKE128 operation.

Parameters:

- **cshake** pointer to the wc_Cshake structure holding state.
- **in** the message data.
- **inLen** length of the message data in bytes.

See:

- wc_InitCshake128
- wc_Cshake128_Final

Return:

- 0 Returned upon successfully adding the data.
- BAD_FUNC_ARG Returned when cshake is NULL, or in is NULL with a non-zero length.

Example

```
wc_Cshake128_Update(&cshake, data, dataLen);
```

C.52.2.65 function wc_Cshake128_Final

```
int wc_Cshake128_Final(
    wc_Cshake * cshake,
    byte * out,
    word32 outLen
)
```

This function finalizes a cSHAKE128 operation, squeezing outLen bytes of output. cSHAKE is an XOF, so a longer squeeze extends a shorter one over the same input.

Parameters:

- **cshake** pointer to the wc_Cshake structure holding state.
- **out** buffer to hold the output.

- **outLen** number of output bytes to produce.

See:

- wc_InitCshake128
- wc_Cshake128_Update

Return:

- 0 Returned upon successfully finalizing.
- BAD_FUNC_ARG Returned when cshake is NULL, or out is NULL with a non-zero length.

Example

```
byte out[32];
wc_Cshake128_Final(&cshake, out, sizeof(out));
```

C.52.2.66 function wc_Cshake128_Free

```
void wc_Cshake128_Free(
    wc_Cshake * cshake
)
```

This function releases any resources associated with a cSHAKE128 operation. Passing NULL is safe and does nothing.

Parameters:

- **cshake** pointer to the wc_Cshake structure to free, may be NULL.

See: wc_InitCshake128

Example

```
wc_Cshake128_Free(&cshake);
```

C.52.2.67 function wc_Cshake128

```
int wc_Cshake128(
    const byte * name,
    word32 nameLen,
    const byte * custom,
    word32 customLen,
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen
)
```

This function computes a cSHAKE128 over a single message in one call, initializing, updating and finalizing internally.

Parameters:

- **name** function-name string, or NULL when nameLen is 0.
- **nameLen** length of the function-name string in bytes.
- **custom** customization string, or NULL when customLen is 0.
- **customLen** length of the customization string in bytes.
- **in** the message data.
- **inLen** length of the message data in bytes.
- **out** buffer to hold the output.
- **outLen** number of output bytes to produce.

See:

- wc_InitCshake128
- wc_Cshake128_Final

Return:

- 0 Returned upon success.
- BAD_FUNC_ARG Returned when a required pointer is NULL with a non-zero length.

Example

```
byte out[32];
wc_Cshake128(NULL, 0, custom, customLen, data, dataLen, out, sizeof(out));
```

C.52.2.68 function wc_InitCshake256

```
int wc_InitCshake256(
    wc_Cshake * cshake,
    const byte * name,
    word32 nameLen,
    const byte * custom,
    word32 customLen,
    void * heap,
    int devId
)
```

This function initializes a cSHAKE256 operation (customizable SHAKE, NIST SP 800-185) with a function-name and customization string. cSHAKE is enabled together with KMAC (WOLFSSL_KMAC). When both strings are empty, cSHAKE reduces to plain SHAKE256.

Parameters:

- **cshake** pointer to the wc_Cshake structure to initialize.
- **name** function-name string (reserved for NIST-defined functions); use an empty string for application customization via custom.
- **nameLen** length of the function-name string in bytes.
- **custom** customization string, or NULL when customLen is 0.
- **customLen** length of the customization string in bytes.
- **heap** pointer to a heap hint, may be NULL.
- **devId** device identifier, or INVALID_DEVID.

See:

- wc_Cshake256_Update
- wc_Cshake256_Final
- wc_Cshake256_Free
- wc_Cshake256

Return:

- 0 Returned upon successfully initializing.
- BAD_FUNC_ARG Returned when a required pointer is NULL.

Example

```
wc_Cshake cshake;
byte out[32];

if (wc_InitCshake256(&cshake, NULL, 0, custom, customLen, NULL,
        INVALID_DEVID) == 0) {
```

```
    wc_Cshake256_Update(&cshake, data, dataLen);
    wc_Cshake256_Final(&cshake, out, sizeof(out));
    wc_Cshake256_Free(&cshake);
}
```

C.52.2.69 function wc_Cshake256_Update

```
int wc_Cshake256_Update(
    wc_Cshake * cshake,
    const byte * in,
    word32 inLen
)
```

This function absorbs message data into a cSHAKE256 operation.

Parameters:

- **cshake** pointer to the wc_Cshake structure holding state.
- **in** the message data.
- **inLen** length of the message data in bytes.

See:

- wc_InitCshake256
- wc_Cshake256_Final

Return:

- 0 Returned upon successfully adding the data.
- BAD_FUNC_ARG Returned when cshake is NULL, or in is NULL with a non-zero length.

Example

```
wc_Cshake256_Update(&cshake, data, dataLen);
```

C.52.2.70 function wc_Cshake256_Final

```
int wc_Cshake256_Final(
    wc_Cshake * cshake,
    byte * out,
    word32 outLen
)
```

This function finalizes a cSHAKE256 operation, squeezing outLen bytes of output. cSHAKE is an XOF, so a longer squeeze extends a shorter one over the same input.

Parameters:

- **cshake** pointer to the wc_Cshake structure holding state.
- **out** buffer to hold the output.
- **outLen** number of output bytes to produce.

See:

- wc_InitCshake256
- wc_Cshake256_Update

Return:

- 0 Returned upon successfully finalizing.
- BAD_FUNC_ARG Returned when cshake is NULL, or out is NULL with a non-zero length.

Example

```
byte out[32];
wc_Cshake256_Final(&cshake, out, sizeof(out));
```

C.52.2.71 function wc_Cshake256_Free

```
void wc_Cshake256_Free(
    wc_Cshake * cshake
)
```

This function releases any resources associated with a cSHAKE256 operation. Passing NULL is safe and does nothing.

Parameters:

- **cshake** pointer to the wc_Cshake structure to free, may be NULL.

See: wc_InitCshake256

Example

```
wc_Cshake256_Free(&cshake);
```

C.52.2.72 function wc_Cshake256

```
int wc_Cshake256(
    const byte * name,
    word32 nameLen,
    const byte * custom,
    word32 customLen,
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen
)
```

This function computes a cSHAKE256 over a single message in one call, initializing, updating and finalizing internally.

Parameters:

- **name** function-name string, or NULL when nameLen is 0.
- **nameLen** length of the function-name string in bytes.
- **custom** customization string, or NULL when customLen is 0.
- **customLen** length of the customization string in bytes.
- **in** the message data.
- **inLen** length of the message data in bytes.
- **out** buffer to hold the output.
- **outLen** number of output bytes to produce.

See:

- wc_InitCshake256
- wc_Cshake256_Final

Return:

- 0 Returned upon success.
- BAD_FUNC_ARG Returned when a required pointer is NULL with a non-zero length.

Example

```
byte out[32];
wc_Cshake256(NULL, 0, custom, customLen, data, dataLen, out, sizeof(out));
```

C.52.2.73 function wc_Cshake128_Copy

```
int wc_Cshake128_Copy(
    wc_Cshake * src,
    wc_Cshake * dst
)
```

This function copies the state of a cSHAKE128 operation so it can be finalized more than once, for example over a common message prefix. dst must already be an initialized wc_Cshake.

Parameters:

- **src** pointer to the wc_Cshake structure to copy from.
- **dst** pointer to the wc_Cshake structure to copy into.

See:

- wc_InitCshake128
- wc_Cshake128_Update
- wc_Cshake128_Final

Return:

- 0 Returned upon success.
- BAD_FUNC_ARG Returned when src or dst is NULL.

Example

```
wc_Cshake cshake;
wc_Cshake copy;

wc_InitCshake128(&cshake, NULL, 0, custom, customLen, NULL, INVALID_DEVID);
wc_InitCshake128(&copy, NULL, 0, custom, customLen, NULL, INVALID_DEVID);
wc_Cshake128_Update(&cshake, prefix, prefixLen);
wc_Cshake128_Copy(&cshake, &copy);
```

C.52.2.74 function wc_Cshake256_Copy

```
int wc_Cshake256_Copy(
    wc_Cshake * src,
    wc_Cshake * dst
)
```

This function copies the state of a cSHAKE256 operation so it can be finalized more than once, for example over a common message prefix. dst must already be an initialized wc_Cshake.

Parameters:

- **src** pointer to the wc_Cshake structure to copy from.
- **dst** pointer to the wc_Cshake structure to copy into.

See:

- wc_InitCshake256
- wc_Cshake256_Update
- wc_Cshake256_Final

Return:

- 0 Returned upon success.
- BAD_FUNC_ARG Returned when src or dst is NULL.

Example

```
wc_Cshake cshake;
wc_Cshake copy;

wc_InitCshake256(&cshake, NULL, 0, custom, customLen, NULL, INVALID_DEVID);
wc_InitCshake256(&copy, NULL, 0, custom, customLen, NULL, INVALID_DEVID);
wc_Cshake256_Update(&cshake, prefix, prefixLen);
wc_Cshake256_Copy(&cshake, &copy);
```

C.52.3 Source code

```
int wc_InitSha3_224(wc_Sha3* sha3, void* heap, int devId);

int wc_Sha3_224_Update(wc_Sha3* sha, const byte* data, word32 len);

int wc_Sha3_224_Final(wc_Sha3* sha3, byte* hash);

void wc_Sha3_224_Free(wc_Sha3* sha3);

        wc_Sha3_224_Update(&sha3, data2, sizeof(data2));
        wc_Sha3_224_Final(&sha3, hash);
        wc_Sha3_224_Free(&sha3);
    }
    \endcode

    \sa wc_InitSha3_224
    \sa wc_Sha3_224_Final
    \sa wc_Sha3_224_Free
*/
int wc_Sha3_224_Reset(wc_Sha3* sha3);

int wc_Sha3_224_GetHash(wc_Sha3* sha3, byte* hash);

int wc_Sha3_224_Copy(wc_Sha3* sha3, wc_Sha3* dst);

int wc_InitSha3_256(wc_Sha3* sha3, void* heap, int devId);

int wc_Sha3_256_Update(wc_Sha3* sha, const byte* data, word32 len);

int wc_Sha3_256_Final(wc_Sha3* sha3, byte* hash);

void wc_Sha3_256_Free(wc_Sha3* sha3);

        wc_Sha3_256_Update(&sha3, data2, sizeof(data2));
        wc_Sha3_256_Final(&sha3, hash);
        wc_Sha3_256_Free(&sha3);
    }
    \endcode

    \sa wc_InitSha3_256
```

```
    \sa wc_Sha3_256_Final
    \sa wc_Sha3_256_Free
*/
int wc_Sha3_256_Reset(wc_Sha3* sha3);

int wc_Sha3_256_GetHash(wc_Sha3* sha3, byte* hash);

int wc_Sha3_256_Copy(wc_Sha3* sha3, wc_Sha3* dst);

int wc_InitSha3_384(wc_Sha3* sha3, void* heap, int devId);

int wc_Sha3_384_Update(wc_Sha3* sha, const byte* data, word32 len);

int wc_Sha3_384_Final(wc_Sha3* sha3, byte* hash);

void wc_Sha3_384_Free(wc_Sha3* sha3);

        wc_Sha3_384_Update(&sha3, data2, sizeof(data2));
        wc_Sha3_384_Final(&sha3, hash);
        wc_Sha3_384_Free(&sha3);
    }
    \endcode

    \sa wc_InitSha3_384
    \sa wc_Sha3_384_Final
    \sa wc_Sha3_384_Free
*/
int wc_Sha3_384_Reset(wc_Sha3* sha3);

int wc_Sha3_384_GetHash(wc_Sha3* sha3, byte* hash);

int wc_Sha3_384_Copy(wc_Sha3* sha3, wc_Sha3* dst);

int wc_InitSha3_512(wc_Sha3* sha3, void* heap, int devId);

int wc_Sha3_512_Update(wc_Sha3* sha, const byte* data, word32 len);

int wc_Sha3_512_Final(wc_Sha3* sha3, byte* hash);

void wc_Sha3_512_Free(wc_Sha3* sha3);

        wc_Sha3_512_Update(&sha3, data2, sizeof(data2));
        wc_Sha3_512_Final(&sha3, hash);
        wc_Sha3_512_Free(&sha3);
    }
    \endcode

    \sa wc_InitSha3_512
    \sa wc_Sha3_512_Final
    \sa wc_Sha3_512_Free
*/
int wc_Sha3_512_Reset(wc_Sha3* sha3);

int wc_Sha3_512_GetHash(wc_Sha3* sha3, byte* hash);
```

```
int wc_Sha3_512_Copy(wc_Sha3* sha3, wc_Sha3* dst);

int wc_InitShake128(wc_Shake* shake, void* heap, int devId);

int wc_Shake128_Update(wc_Shake* sha, const byte* data, word32 len);

int wc_Shake128_Final(wc_Shake* shake, byte* hash, word32 hashLen);

int wc_Shake128_Absorb(wc_Shake* sha, const byte* data, word32 len);

int wc_Shake128_SqueezeBlocks(wc_Shake* shake, byte* out, word32 blockCnt);

void wc_Shake128_Free(wc_Shake* shake);

        wc_Shake128_Update(&shake, data2, sizeof(data2));
        wc_Shake128_Final(&shake, hash, sizeof(hash));
        wc_Shake128_Free(&shake);
    }
    \endcode

    \sa wc_InitShake128
    \sa wc_Shake128_Final
    \sa wc_Shake128_Free
*/
int wc_Shake128_Reset(wc_Shake* shake);

int wc_Shake128_GetHash(wc_Shake* shake, byte* hash);

int wc_Shake128_Copy(wc_Shake* src, wc_Sha3* dst);

int wc_InitShake256(wc_Shake* shake, void* heap, int devId);

int wc_Shake256_Update(wc_Shake* sha, const byte* data, word32 len);

int wc_Shake256_Final(wc_Shake* shake, byte* hash, word32 hashLen);

int wc_Shake256_Absorb(wc_Shake* sha, const byte* data, word32 len);

int wc_Shake256_SqueezeBlocks(wc_Shake* shake, byte* out, word32 blockCnt);

void wc_Shake256_Free(wc_Shake* shake);

        wc_Shake256_Update(&shake, data2, sizeof(data2));
        wc_Shake256_Final(&shake, hash, sizeof(hash));
        wc_Shake256_Free(&shake);
    }
    \endcode

    \sa wc_InitShake256
    \sa wc_Shake256_Final
    \sa wc_Shake256_Free
*/
int wc_Shake256_Reset(wc_Shake* shake);
```

```
int wc_Shake256_GetHash(wc_Shake* shake, byte* hash);

int wc_Shake256_Copy(wc_Shake* src, wc_Sha3* dst);

int wc_InitKmac128(wc_Kmac* kmac, const byte* key, word32 keyLen,
    const byte* custom, word32 customLen, void* heap, int devId);

int wc_Kmac128_Update(wc_Kmac* kmac, const byte* in, word32 inLen);

int wc_Kmac128_Final(wc_Kmac* kmac, byte* out, word32 outLen);

int wc_Kmac128_FinalXof(wc_Kmac* kmac, byte* out, word32 outLen);

void wc_Kmac128_Free(wc_Kmac* kmac);

int wc_Kmac128Hash(const byte* key, word32 keyLen, const byte* custom,
    word32 customLen, const byte* in, word32 inLen, byte* out, word32 outLen);

int wc_Kmac128HashXof(const byte* key, word32 keyLen, const byte* custom,
    word32 customLen, const byte* in, word32 inLen, byte* out, word32 outLen);

int wc_InitKmac256(wc_Kmac* kmac, const byte* key, word32 keyLen,
    const byte* custom, word32 customLen, void* heap, int devId);

int wc_Kmac256_Update(wc_Kmac* kmac, const byte* in, word32 inLen);

int wc_Kmac256_Final(wc_Kmac* kmac, byte* out, word32 outLen);

int wc_Kmac256_FinalXof(wc_Kmac* kmac, byte* out, word32 outLen);

void wc_Kmac256_Free(wc_Kmac* kmac);

int wc_Kmac256Hash(const byte* key, word32 keyLen, const byte* custom,
    word32 customLen, const byte* in, word32 inLen, byte* out, word32 outLen);

int wc_Kmac256HashXof(const byte* key, word32 keyLen, const byte* custom,
    word32 customLen, const byte* in, word32 inLen, byte* out, word32 outLen);


int wc_Kmac128_Copy(wc_Kmac* src, wc_Kmac* dst);

int wc_Kmac256_Copy(wc_Kmac* src, wc_Kmac* dst);

int wc_InitCshake128(wc_Cshake* cshake, const byte* name, word32 nameLen,
    const byte* custom, word32 customLen, void* heap, int devId);

int wc_Cshake128_Update(wc_Cshake* cshake, const byte* in, word32 inLen);

int wc_Cshake128_Final(wc_Cshake* cshake, byte* out, word32 outLen);

void wc_Cshake128_Free(wc_Cshake* cshake);

int wc_Cshake128(const byte* name, word32 nameLen, const byte* custom,
```

```
    word32 customLen, const byte* in, word32 inLen, byte* out, word32 outLen);

int wc_InitCshake256(wc_Cshake* cshake, const byte* name, word32 nameLen,
    const byte* custom, word32 customLen, void* heap, int devId);

int wc_Cshake256_Update(wc_Cshake* cshake, const byte* in, word32 inLen);

int wc_Cshake256_Final(wc_Cshake* cshake, byte* out, word32 outLen);

void wc_Cshake256_Free(wc_Cshake* cshake);

int wc_Cshake256(const byte* name, word32 nameLen, const byte* custom,
    word32 customLen, const byte* in, word32 inLen, byte* out, word32 outLen);


int wc_Cshake128_Copy(wc_Cshake* src, wc_Cshake* dst);

int wc_Cshake256_Copy(wc_Cshake* src, wc_Cshake* dst);
```

C.53 sha512.h

C.53.1 Functions

	Name
int	**wc_InitSha512**(wc_Sha512 * sha)This function initializes SHA512. This is automatically called by wc_Sha512Hash.
int	**wc_Sha512Update**(wc_Sha512 * sha, const byte * data, word32 len)Can be called to continually hash the provided byte array of length len.
int	**wc_Sha512Final**(wc_Sha512 * sha512, byte * hash)Finalizes hashing of data. Result is placed into hash.
int	**wc_InitSha384**(wc_Sha384 * sha)This function initializes SHA384. This is automatically called by wc_Sha384Hash.
int	**wc_Sha384Update**(wc_Sha384 * sha, const byte * data, word32 len)Can be called to continually hash the provided byte array of length len.
int	**wc_Sha384Final**(wc_Sha384 * sha384, byte * hash)Finalizes hashing of data. Result is placed into hash.
int	**wc_InitSha512_ex**(wc_Sha512 * sha, void * heap, int devId)Initializes SHA512 with heap and device ID.
int	**wc_Sha512FinalRaw**(wc_Sha512 * sha512, byte * hash)Gets raw hash without finalizing.
void	**wc_Sha512Free**(wc_Sha512 * sha)Frees SHA512 resources.

	Name
int	**wc_Sha512Reset**(wc_Sha512 * sha)Resets the sha structure to its freshly initialized state, continuing with its established allocations, heap hint, and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha512Free for teardown at end of life.
int	**wc_Sha512GetHash**(wc_Sha512 * sha512, byte * hash)Gets SHA512 hash without finalizing.
int	**wc_Sha512Copy**(wc_Sha512 * src, wc_Sha512 * dst)Copies SHA512 context.
int	**wc_Sha512_Grow**(wc_Sha512 * sha512, const byte * in, int inSz)Grows SHA512 buffer with input data.
int	**wc_Sha512SetFlags**(wc_Sha512 * sha512, word32 flags)Sets SHA512 flags.
int	**wc_Sha512GetFlags**(wc_Sha512 * sha512, word32 * flags)Gets SHA512 flags.
int	**wc_Sha512Transform**(wc_Sha512 * sha, const unsigned char * data)Transforms SHA512 block.
int	**wc_InitSha512_224**(wc_Sha512 * sha)Initializes SHA512/224.
int	**wc_InitSha512_224_ex**(wc_Sha512 * sha, void * heap, int devId)Initializes SHA512/224 with heap and device ID.
int	**wc_Sha512_224Update**(wc_Sha512 * sha, const byte * data, word32 len)Updates SHA512/224 hash with data.
int	**wc_Sha512_224FinalRaw**(wc_Sha512 * sha512, byte * hash)Gets raw SHA512/224 hash without finalizing.
int	**wc_Sha512_224Final**(wc_Sha512 * sha512, byte * hash)Finalizes SHA512/224 hash.
void	**wc_Sha512_224Free**(wc_Sha512 * sha)Frees SHA512/224 resources.
int	**wc_Sha512_224Reset**(wc_Sha512 * sha)Resets the sha structure to its freshly initialized state, continuing with its established allocations, heap hint, and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha512_224Free for teardown at end of life.
int	**wc_Sha512_224GetHash**(wc_Sha512 * sha512, byte * hash)Gets SHA512/224 hash without finalizing.

	Name
int	**wc_Sha512_224Copy**(wc_Sha512 * src, wc_Sha512 * dst)Copies SHA512/224 context.
int	**wc_Sha512_224SetFlags**(wc_Sha512 * sha512, word32 flags)Sets SHA512/224 flags.
int	**wc_Sha512_224GetFlags**(wc_Sha512 * sha512, word32 * flags)Gets SHA512/224 flags.
int	**wc_Sha512_224Transform**(wc_Sha512 * sha, const unsigned char * data)Transforms SHA512/224 block.
int	**wc_InitSha512_256**(wc_Sha512 * sha)Initializes SHA512/256.
int	**wc_InitSha512_256_ex**(wc_Sha512 * sha, void * heap, int devId)Initializes SHA512/256 with heap and device ID.
int	**wc_Sha512_256Update**(wc_Sha512 * sha, const byte * data, word32 len)Updates SHA512/256 hash with data.
int	**wc_Sha512_256FinalRaw**(wc_Sha512 * sha512, byte * hash)Gets raw SHA512/256 hash without finalizing.
int	**wc_Sha512_256Final**(wc_Sha512 * sha512, byte * hash)Finalizes SHA512/256 hash.
void	**wc_Sha512_256Free**(wc_Sha512 * sha)Frees SHA512/256 resources.
int	**wc_Sha512_256Reset**(wc_Sha512 * sha)Resets the sha structure to its freshly initialized state, continuing with its established allocations, heap hint, and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha512_256Free for teardown at end of life.
int	**wc_Sha512_256GetHash**(wc_Sha512 * sha512, byte * hash)Gets SHA512/256 hash without finalizing.
int	**wc_Sha512_256Copy**(wc_Sha512 * src, wc_Sha512 * dst)Copies SHA512/256 context.
int	**wc_Sha512_256SetFlags**(wc_Sha512 * sha512, word32 flags)Sets SHA512/256 flags.
int	**wc_Sha512_256GetFlags**(wc_Sha512 * sha512, word32 * flags)Gets SHA512/256 flags.
int	**wc_Sha512_256Transform**(wc_Sha512 * sha, const unsigned char * data)Transforms SHA512/256 block.
int	**wc_InitSha384_ex**(wc_Sha384 * sha, void * heap, int devId)Initializes SHA384 with heap and device ID.
int	**wc_Sha384FinalRaw**(wc_Sha384 * sha384, byte * hash)Gets raw SHA384 hash without finalizing.

	Name
void	**wc_Sha384Free**(wc_Sha384 * sha)Frees SHA384 resources.
int	**wc_Sha384Reset**(wc_Sha384 * sha)Resets the sha structure to its freshly initialized state, continuing with its established allocations, heap hint, and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha384Free for teardown at end of life.
int	**wc_Sha384GetHash**(wc_Sha384 * sha384, byte * hash)Gets SHA384 hash without finalizing.
int	**wc_Sha384Copy**(wc_Sha384 * src, wc_Sha384 * dst)Copies SHA384 context.
int	**wc_Sha384_Grow**(wc_Sha384 * sha384, const byte * in, int inSz)Grows SHA384 buffer with input data.
int	**wc_Sha384SetFlags**(wc_Sha384 * sha384, word32 flags)Sets SHA384 flags.
int	**wc_Sha384GetFlags**(wc_Sha384 * sha384, word32 * flags)Gets SHA384 flags.
int	**wc_Sha384Transform**(wc_Sha384 * sha, const unsigned char * data)Transforms SHA384 block.

C.53.2 Functions Documentation

C.53.2.1 function wc_InitSha512

```
int wc_InitSha512(
    wc_Sha512 * sha
)
```

This function initializes SHA512. This is automatically called by wc_Sha512Hash.

Parameters:

- **sha512** pointer to the sha512 structure to use for encryption

See:

- wc_Sha512Hash
- wc_Sha512Update
- wc_Sha512Final

Return: 0 Returned upon successfully initializing

Example

```
Sha512 sha512[1];
if ((ret = wc_InitSha512(sha512)) != 0) {
   WOLFSSL_MSG("wc_InitSha512 failed");
}
else {
```

```
   wc_Sha512Update(sha512, data, len);
   wc_Sha512Final(sha512, hash);
}
```

C.53.2.2 function wc_Sha512Update

```
int wc_Sha512Update(
    wc_Sha512 * sha,
    const byte * data,
    word32 len
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **sha512** pointer to the sha512 structure to use for encryption
- **data** the data to be hashed
- **len** length of data to be hashed

See:

- wc_Sha512Hash
- wc_Sha512Final
- wc_InitSha512

Return: 0 Returned upon successfully adding the data to the digest.

Example

```
Sha512 sha512[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha512(sha512)) != 0) {
   WOLFSSL_MSG("wc_InitSha512 failed");
}
else {
   wc_Sha512Update(sha512, data, len);
   wc_Sha512Final(sha512, hash);
}
```

C.53.2.3 function wc_Sha512Final

```
int wc_Sha512Final(
    wc_Sha512 * sha512,
    byte * hash
)
```

Finalizes hashing of data. Result is placed into hash.

Parameters:

- **sha512** pointer to the sha512 structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Sha512Hash
- wc_Sha512Final
- wc_InitSha512

Return: 0 Returned upon successfully finalizing the hash.

Example

```
Sha512 sha512[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha512(sha512)) != 0) {
    WOLFSSL_MSG("wc_InitSha512 failed");
}
else {
   wc_Sha512Update(sha512, data, len);
   wc_Sha512Final(sha512, hash);
}
```

C.53.2.4 function wc_InitSha384

```
int wc_InitSha384(
    wc_Sha384 * sha
)
```

This function initializes SHA384. This is automatically called by wc_Sha384Hash.

Parameters:

- **sha384** pointer to the sha384 structure to use for encryption

See:

- wc_Sha384Hash
- wc_Sha384Update
- wc_Sha384Final

Return: 0 Returned upon successfully initializing

Example

```
Sha384 sha384[1];
if ((ret = wc_InitSha384(sha384)) != 0) {
   WOLFSSL_MSG("wc_InitSha384 failed");
}
else {
   wc_Sha384Update(sha384, data, len);
   wc_Sha384Final(sha384, hash);
}
```

C.53.2.5 function wc_Sha384Update

```
int wc_Sha384Update(
    wc_Sha384 * sha,
    const byte * data,
    word32 len
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **sha384** pointer to the sha384 structure to use for encryption
- **data** the data to be hashed

- **len** length of data to be hashed

See:

- wc_Sha384Hash
- wc_Sha384Final
- wc_InitSha384

Return: 0 Returned upon successfully adding the data to the digest.

Example

```
Sha384 sha384[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha384(sha384)) != 0) {
   WOLFSSL_MSG("wc_InitSha384 failed");
}
else {
   wc_Sha384Update(sha384, data, len);
   wc_Sha384Final(sha384, hash);
}
```

C.53.2.6 function wc_Sha384Final

```
int wc_Sha384Final(
    wc_Sha384 * sha384,
    byte * hash
)
```

Finalizes hashing of data. Result is placed into hash.

Parameters:

- **sha384** pointer to the sha384 structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Sha384Hash
- wc_Sha384Final
- wc_InitSha384

Return: 0 Returned upon successfully finalizing.

Example

```
Sha384 sha384[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha384(sha384)) != 0) {
   WOLFSSL_MSG("wc_InitSha384 failed");
}
else {
   wc_Sha384Update(sha384, data, len);
   wc_Sha384Final(sha384, hash);
}
```

C.53.2.7 function wc_InitSha512_ex

```
int wc_InitSha512_ex(
    wc_Sha512 * sha,
    void * heap,
    int devId
)
```

Initializes SHA512 with heap and device ID.

Parameters:

- **sha** SHA512 structure
- **heap** Heap hint
- **devId** Device ID

See: wc_InitSha512

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
int ret = wc_InitSha512_ex(&sha, NULL, INVALID_DEVID);
```

C.53.2.8 function wc_Sha512FinalRaw

```
int wc_Sha512FinalRaw(
    wc_Sha512 * sha512,
    byte * hash
)
```

Gets raw hash without finalizing.

Parameters:

- **sha512** SHA512 structure
- **hash** Output hash buffer

See: wc_Sha512Final

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
byte hash[WC_SHA512_DIGEST_SIZE];
int ret = wc_Sha512FinalRaw(&sha, hash);
```

C.53.2.9 function wc_Sha512Free

```
void wc_Sha512Free(
    wc_Sha512 * sha
)
```

Frees SHA512 resources.

Parameters:

- **sha** SHA512 structure

See: wc_InitSha512

Return: none No returns

Example

```
wc_Sha512 sha;
wc_InitSha512(&sha);
wc_Sha512Free(&sha);
```

C.53.2.10 function wc_Sha512Reset

```
int wc_Sha512Reset(
    wc_Sha512 * sha
)
```

Resets the sha structure to its freshly initialized state, continuing with its established allocations, heap hint, and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha512Free for teardown at end of life.

Parameters:

- **sha** Pointer to the sha structure to be reset.

See:

- wc_InitSha512
- wc_Sha512Final
- wc_Sha512Free

Return:

- 0 Returned upon successfully resetting the structure.
- BAD_FUNC_ARG Returned if sha is NULL.
- negative Other values may be returned on reinitialization failure in fallback (non-plain-software) configurations.

Example

```
wc_Sha512 sha;
byte data[] = { Data to be discarded };
byte data2[] = { Data to be hashed };
byte hash[WC_SHA512_DIGEST_SIZE];

if ((ret = wc_InitSha512(&sha)) == 0) {
    wc_Sha512Update(&sha, data, sizeof(data));
    wc_Sha512Reset(&sha); /* discard the partially hashed message */
    wc_Sha512Update(&sha, data2, sizeof(data2));
    wc_Sha512Final(&sha, hash);
    wc_Sha512Free(&sha);
}
```

C.53.2.11 function wc_Sha512GetHash

```
int wc_Sha512GetHash(
    wc_Sha512 * sha512,
    byte * hash
)
```

Gets SHA512 hash without finalizing.

Parameters:

- **sha512** SHA512 structure
- **hash** Output hash buffer

See: wc_Sha512Final

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
byte hash[WC_SHA512_DIGEST_SIZE];
int ret = wc_Sha512GetHash(&sha, hash);
```

C.53.2.12 function wc_Sha512Copy

```
int wc_Sha512Copy(
    wc_Sha512 * src,
    wc_Sha512 * dst
)
```

Copies SHA512 context.

Parameters:

- **src** Source SHA512 structure
- **dst** Destination SHA512 structure (must be zeroed or previously initialized)

See: wc_InitSha512

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 src, dst;
memset(&dst, 0, sizeof(dst));
int ret = wc_InitSha512(&src);
if (ret == 0)
    ret = wc_Sha512Copy(&src, &dst);
```

C.53.2.13 function wc_Sha512_Grow

```
int wc_Sha512_Grow(
    wc_Sha512 * sha512,
    const byte * in,
    int inSz
)
```

Grows SHA512 buffer with input data.

Parameters:

- **sha512** SHA512 structure
- **in** Input data

- **inSz** Input size

See: wc_Sha512Update

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
byte data[100];
int ret = wc_Sha512_Grow(&sha, data, sizeof(data));
```

C.53.2.14 function wc_Sha512SetFlags

```
int wc_Sha512SetFlags(
    wc_Sha512 * sha512,
    word32 flags
)
```

Sets SHA512 flags.

Parameters:

- **sha512** SHA512 structure
- **flags** Flags to set

See: wc_InitSha512

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
int ret = wc_Sha512SetFlags(&sha, WC_HASH_FLAG_WILLCOPY);
```

C.53.2.15 function wc_Sha512GetFlags

```
int wc_Sha512GetFlags(
    wc_Sha512 * sha512,
    word32 * flags
)
```

Gets SHA512 flags.

Parameters:

- **sha512** SHA512 structure
- **flags** Pointer to store flags

See: wc_Sha512SetFlags

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
word32 flags;
int ret = wc_Sha512GetFlags(&sha, &flags);
```

C.53.2.16 function wc_Sha512Transform

```
int wc_Sha512Transform(
    wc_Sha512 * sha,
    const unsigned char * data
)
```

Transforms SHA512 block.

Parameters:

- **sha** SHA512 structure
- **data** Block data

See: wc_Sha512Update

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
unsigned char block[WC_SHA512_BLOCK_SIZE];
int ret = wc_Sha512Transform(&sha, block);
```

C.53.2.17 function wc_InitSha512_224

```
int wc_InitSha512_224(
    wc_Sha512 * sha
)
```

Initializes SHA512/224.

Parameters:

- **sha** SHA512 structure

See: wc_Sha512_224Update

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
int ret = wc_InitSha512_224(&sha);
```

C.53.2.18 function wc_InitSha512_224_ex

```
int wc_InitSha512_224_ex(
    wc_Sha512 * sha,
    void * heap,
    int devId
)
```

Initializes SHA512/224 with heap and device ID.

Parameters:

- **sha** SHA512 structure
- **heap** Heap hint
- **devId** Device ID

See: wc_InitSha512_224

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
int ret = wc_InitSha512_224_ex(&sha, NULL, INVALID_DEVID);
```

C.53.2.19 function wc_Sha512_224Update

```
int wc_Sha512_224Update(
    wc_Sha512 * sha,
    const byte * data,
    word32 len
)
```

Updates SHA512/224 hash with data.

Parameters:

- **sha** SHA512 structure
- **data** Input data
- **len** Input size

See: wc_InitSha512_224

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
byte data[100];
int ret = wc_Sha512_224Update(&sha, data, sizeof(data));
```

C.53.2.20 function wc_Sha512_224FinalRaw

```
int wc_Sha512_224FinalRaw(
    wc_Sha512 * sha512,
    byte * hash
)
```

Gets raw SHA512/224 hash without finalizing.

Parameters:

- **sha512** SHA512 structure
- **hash** Output hash buffer

See: wc_Sha512_224Final

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
byte hash[WC_SHA512_224_DIGEST_SIZE];
int ret = wc_Sha512_224FinalRaw(&sha, hash);
```

C.53.2.21 function wc_Sha512_224Final

```
int wc_Sha512_224Final(
    wc_Sha512 * sha512,
    byte * hash
)
```

Finalizes SHA512/224 hash.

Parameters:

- **sha512** SHA512 structure
- **hash** Output hash buffer

See: wc_Sha512_224Update

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
byte hash[WC_SHA512_224_DIGEST_SIZE];
int ret = wc_Sha512_224Final(&sha, hash);
```

C.53.2.22 function wc_Sha512_224Free

```
void wc_Sha512_224Free(
    wc_Sha512 * sha
)
```

Frees SHA512/224 resources.

Parameters:

- **sha** SHA512 structure

See: wc_InitSha512_224

Return: none No returns

Example

```
wc_Sha512 sha;
wc_InitSha512_224(&sha);
wc_Sha512_224Free(&sha);
```

C.53.2.23 function wc_Sha512_224Reset

```
int wc_Sha512_224Reset(
    wc_Sha512 * sha
)
```

Resets the sha structure to its freshly initialized state, continuing with its established allocations, heap hint, and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha512_224Free for teardown at end of life.

Parameters:

- **sha** Pointer to the sha structure to be reset.

See:

- wc_InitSha512_224
- wc_Sha512_224Final
- wc_Sha512_224Free

Return:

- 0 Returned upon successfully resetting the structure.
- BAD_FUNC_ARG Returned if sha is NULL.
- negative Other values may be returned on reinitialization failure in fallback (non-plain-software) configurations.

Example

```
wc_Sha512 sha;
byte data[] = { Data to be discarded };
byte data2[] = { Data to be hashed };
byte hash[WC_SHA512_224_DIGEST_SIZE];

if ((ret = wc_InitSha512_224(&sha)) == 0) {
    wc_Sha512_224Update(&sha, data, sizeof(data));
    wc_Sha512_224Reset(&sha); /* discard the partially hashed message */
    wc_Sha512_224Update(&sha, data2, sizeof(data2));
    wc_Sha512_224Final(&sha, hash);
    wc_Sha512_224Free(&sha);
}
```

C.53.2.24 function wc_Sha512_224GetHash

```
int wc_Sha512_224GetHash(
    wc_Sha512 * sha512,
    byte * hash
)
```

Gets SHA512/224 hash without finalizing.

Parameters:

- **sha512** SHA512 structure
- **hash** Output hash buffer

See: wc_Sha512_224Final

Return:

- 0 on success

- negative on error

Example

```
wc_Sha512 sha;
byte hash[WC_SHA512_224_DIGEST_SIZE];
int ret = wc_Sha512_224GetHash(&sha, hash);
```

C.53.2.25 function wc_Sha512_224Copy

```
int wc_Sha512_224Copy(
    wc_Sha512 * src,
    wc_Sha512 * dst
)
```

Copies SHA512/224 context.

Parameters:

- **src** Source SHA512 structure
- **dst** Destination SHA512 structure (must be zeroed or previously initialized)

See: wc_InitSha512_224

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 src, dst;
memset(&dst, 0, sizeof(dst));
int ret = wc_InitSha512_224(&src);
if (ret == 0)
    ret = wc_Sha512_224Copy(&src, &dst);
```

C.53.2.26 function wc_Sha512_224SetFlags

```
int wc_Sha512_224SetFlags(
    wc_Sha512 * sha512,
    word32 flags
)
```

Sets SHA512/224 flags.

Parameters:

- **sha512** SHA512 structure
- **flags** Flags to set

See: wc_InitSha512_224

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
int ret = wc_Sha512_224SetFlags(&sha, WC_HASH_FLAG_WILLCOPY);
```

C.53.2.27 function wc_Sha512_224GetFlags

```
int wc_Sha512_224GetFlags(
    wc_Sha512 * sha512,
    word32 * flags
)
```

Gets SHA512/224 flags.

Parameters:

- **sha512** SHA512 structure
- **flags** Pointer to store flags

See: wc_Sha512_224SetFlags

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
word32 flags;
int ret = wc_Sha512_224GetFlags(&sha, &flags);
```

C.53.2.28 function wc_Sha512_224Transform

```
int wc_Sha512_224Transform(
    wc_Sha512 * sha,
    const unsigned char * data
)
```

Transforms SHA512/224 block.

Parameters:

- **sha** SHA512 structure
- **data** Block data

See: wc_Sha512_224Update

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
unsigned char block[WC_SHA512_BLOCK_SIZE];
int ret = wc_Sha512_224Transform(&sha, block);
```

C.53.2.29 function wc_InitSha512_256

```
int wc_InitSha512_256(
    wc_Sha512 * sha
)
```

Initializes SHA512/256.

Parameters:

- **sha** SHA512 structure

See: wc_Sha512_256Update

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
int ret = wc_InitSha512_256(&sha);
```

C.53.2.30 function wc_InitSha512_256_ex

```
int wc_InitSha512_256_ex(
    wc_Sha512 * sha,
    void * heap,
    int devId
)
```

Initializes SHA512/256 with heap and device ID.

Parameters:

- **sha** SHA512 structure
- **heap** Heap hint
- **devId** Device ID

See: wc_InitSha512_256

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
int ret = wc_InitSha512_256_ex(&sha, NULL, INVALID_DEVID);
```

C.53.2.31 function wc_Sha512_256Update

```
int wc_Sha512_256Update(
    wc_Sha512 * sha,
    const byte * data,
    word32 len
)
```

Updates SHA512/256 hash with data.

Parameters:

- **sha** SHA512 structure
- **data** Input data
- **len** Input size

See: wc_InitSha512_256

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
byte data[100];
int ret = wc_Sha512_256Update(&sha, data, sizeof(data));
```

C.53.2.32 function wc_Sha512_256FinalRaw

```
int wc_Sha512_256FinalRaw(
    wc_Sha512 * sha512,
    byte * hash
)
```

Gets raw SHA512/256 hash without finalizing.

Parameters:

- **sha512** SHA512 structure
- **hash** Output hash buffer

See: wc_Sha512_256Final

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
byte hash[WC_SHA512_256_DIGEST_SIZE];
int ret = wc_Sha512_256FinalRaw(&sha, hash);
```

C.53.2.33 function wc_Sha512_256Final

```
int wc_Sha512_256Final(
    wc_Sha512 * sha512,
    byte * hash
)
```

Finalizes SHA512/256 hash.

Parameters:

- **sha512** SHA512 structure
- **hash** Output hash buffer

See: wc_Sha512_256Update

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
byte hash[WC_SHA512_256_DIGEST_SIZE];
int ret = wc_Sha512_256Final(&sha, hash);
```

C.53.2.34 function wc_Sha512_256Free

```
void wc_Sha512_256Free(
    wc_Sha512 * sha
)
```

Frees SHA512/256 resources.

Parameters:

- **sha** SHA512 structure

See: wc_InitSha512_256

Return: none No returns

Example

```
wc_Sha512 sha;
wc_InitSha512_256(&sha);
wc_Sha512_256Free(&sha);
```

C.53.2.35 function wc_Sha512_256Reset

```
int wc_Sha512_256Reset(
    wc_Sha512 * sha
)
```

Resets the sha structure to its freshly initialized state, continuing with its established allocations, heap hint, and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha512_256Free for teardown at end of life.

Parameters:

- **sha** Pointer to the sha structure to be reset.

See:

- wc_InitSha512_256
- wc_Sha512_256Final
- wc_Sha512_256Free

Return:

- 0 Returned upon successfully resetting the structure.
- BAD_FUNC_ARG Returned if sha is NULL.
- negative Other values may be returned on reinitialization failure in fallback (non-plain-software) configurations.

Example

```
wc_Sha512 sha;
byte data[] = { Data to be discarded };
byte data2[] = { Data to be hashed };
byte hash[WC_SHA512_256_DIGEST_SIZE];

if ((ret = wc_InitSha512_256(&sha)) == 0) {
    wc_Sha512_256Update(&sha, data, sizeof(data));
    wc_Sha512_256Reset(&sha); /* discard the partially hashed message */
    wc_Sha512_256Update(&sha, data2, sizeof(data2));
    wc_Sha512_256Final(&sha, hash);
```

```
    wc_Sha512_256Free(&sha);
}
```

C.53.2.36 function wc_Sha512_256GetHash

```
int wc_Sha512_256GetHash(
    wc_Sha512 * sha512,
    byte * hash
)
```

Gets SHA512/256 hash without finalizing.

Parameters:

- **sha512** SHA512 structure
- **hash** Output hash buffer

See: wc_Sha512_256Final

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
byte hash[WC_SHA512_256_DIGEST_SIZE];
int ret = wc_Sha512_256GetHash(&sha, hash);
```

C.53.2.37 function wc_Sha512_256Copy

```
int wc_Sha512_256Copy(
    wc_Sha512 * src,
    wc_Sha512 * dst
)
```

Copies SHA512/256 context.

Parameters:

- **src** Source SHA512 structure
- **dst** Destination SHA512 structure (must be zeroed or previously initialized)

See: wc_InitSha512_256

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 src, dst;
memset(&dst, 0, sizeof(dst));
int ret = wc_InitSha512_256(&src);
if (ret == 0)
    ret = wc_Sha512_256Copy(&src, &dst);
```

C.53.2.38 function wc_Sha512_256SetFlags

```
int wc_Sha512_256SetFlags(
    wc_Sha512 * sha512,
    word32 flags
)
```

Sets SHA512/256 flags.

Parameters:

- **sha512** SHA512 structure
- **flags** Flags to set

See: wc_InitSha512_256

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
int ret = wc_Sha512_256SetFlags(&sha, WC_HASH_FLAG_WILLCOPY);
```

C.53.2.39 function wc_Sha512_256GetFlags

```
int wc_Sha512_256GetFlags(
    wc_Sha512 * sha512,
    word32 * flags
)
```

Gets SHA512/256 flags.

Parameters:

- **sha512** SHA512 structure
- **flags** Pointer to store flags

See: wc_Sha512_256SetFlags

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
word32 flags;
int ret = wc_Sha512_256GetFlags(&sha, &flags);
```

C.53.2.40 function wc_Sha512_256Transform

```
int wc_Sha512_256Transform(
    wc_Sha512 * sha,
    const unsigned char * data
)
```

Transforms SHA512/256 block.

Parameters:

- **sha** SHA512 structure
- **data** Block data

See: wc_Sha512_256Update

Return:

- 0 on success
- negative on error

Example

```
wc_Sha512 sha;
unsigned char block[WC_SHA512_BLOCK_SIZE];
int ret = wc_Sha512_256Transform(&sha, block);
```

C.53.2.41 function wc_InitSha384_ex

```
int wc_InitSha384_ex(
    wc_Sha384 * sha,
    void * heap,
    int devId
)
```

Initializes SHA384 with heap and device ID.

Parameters:

- **sha** SHA384 structure
- **heap** Heap hint
- **devId** Device ID

See: wc_InitSha384

Return:

- 0 on success
- negative on error

Example

```
wc_Sha384 sha;
int ret = wc_InitSha384_ex(&sha, NULL, INVALID_DEVID);
```

C.53.2.42 function wc_Sha384FinalRaw

```
int wc_Sha384FinalRaw(
    wc_Sha384 * sha384,
    byte * hash
)
```

Gets raw SHA384 hash without finalizing.

Parameters:

- **sha384** SHA384 structure
- **hash** Output hash buffer

See: wc_Sha384Final

Return:

- 0 on success
- negative on error

Example

```
wc_Sha384 sha;
byte hash[WC_SHA384_DIGEST_SIZE];
int ret = wc_Sha384FinalRaw(&sha, hash);
```

C.53.2.43 function wc_Sha384Free

```
void wc_Sha384Free(
    wc_Sha384 * sha
)
```

Frees SHA384 resources.

Parameters:

- **sha** SHA384 structure

See: wc_InitSha384

Return: none No returns

Example

```
wc_Sha384 sha;
wc_InitSha384(&sha);
wc_Sha384Free(&sha);
```

C.53.2.44 function wc_Sha384Reset

```
int wc_Sha384Reset(
    wc_Sha384 * sha
)
```

Resets the sha structure to its freshly initialized state, continuing with its established allocations, heap hint, and device association. Equivalent to a fresh init for all subsequent operations; useful for abandoning a partially hashed message, including after a failed Update or Final. Like the Final functions, Reset does not destroy sensitive internal state; use wc_Sha384Free for teardown at end of life.

Parameters:

- **sha** Pointer to the sha structure to be reset.

See:

- wc_InitSha384
- wc_Sha384Final
- wc_Sha384Free

Return:

- 0 Returned upon successfully resetting the structure.
- BAD_FUNC_ARG Returned if sha is NULL.
- negative Other values may be returned on reinitialization failure in fallback (non-plain-software) configurations.

Example

```
wc_Sha384 sha;
byte data[] = { Data to be discarded };
byte data2[] = { Data to be hashed };
byte hash[WC_SHA384_DIGEST_SIZE];
```

```
if ((ret = wc_InitSha384(&sha)) == 0) {
    wc_Sha384Update(&sha, data, sizeof(data));
    wc_Sha384Reset(&sha); /* discard the partially hashed message */
    wc_Sha384Update(&sha, data2, sizeof(data2));
    wc_Sha384Final(&sha, hash);
    wc_Sha384Free(&sha);
}
```

C.53.2.45 function wc_Sha384GetHash

```
int wc_Sha384GetHash(
    wc_Sha384 * sha384,
    byte * hash
)
```

Gets SHA384 hash without finalizing.

Parameters:

- **sha384** SHA384 structure
- **hash** Output hash buffer

See: wc_Sha384Final

Return:

- 0 on success
- negative on error

Example

```
wc_Sha384 sha;
byte hash[WC_SHA384_DIGEST_SIZE];
int ret = wc_Sha384GetHash(&sha, hash);
```

C.53.2.46 function wc_Sha384Copy

```
int wc_Sha384Copy(
    wc_Sha384 * src,
    wc_Sha384 * dst
)
```

Copies SHA384 context.

Parameters:

- **src** Source SHA384 structure
- **dst** Destination SHA384 structure (must be zeroed or previously initialized)

See: wc_InitSha384

Return:

- 0 on success
- negative on error

Example

```
wc_Sha384 src, dst;
memset(&dst, 0, sizeof(dst));
int ret = wc_InitSha384(&src);
```

```
if (ret == 0)
    ret = wc_Sha384Copy(&src, &dst);
```

C.53.2.47 function wc_Sha384_Grow

```
int wc_Sha384_Grow(
    wc_Sha384 * sha384,
    const byte * in,
    int inSz
)
```

Grows SHA384 buffer with input data.

Parameters:

- **sha384** SHA384 structure
- **in** Input data
- **inSz** Input size

See: wc_Sha384Update

Return:

- 0 on success
- negative on error

Example

```
wc_Sha384 sha;
byte data[100];
int ret = wc_Sha384_Grow(&sha, data, sizeof(data));
```

C.53.2.48 function wc_Sha384SetFlags

```
int wc_Sha384SetFlags(
    wc_Sha384 * sha384,
    word32 flags
)
```

Sets SHA384 flags.

Parameters:

- **sha384** SHA384 structure
- **flags** Flags to set

See: wc_InitSha384

Return:

- 0 on success
- negative on error

Example

```
wc_Sha384 sha;
int ret = wc_Sha384SetFlags(&sha, WC_HASH_FLAG_WILLCOPY);
```

C.53.2.49 function wc_Sha384GetFlags

```
int wc_Sha384GetFlags(
    wc_Sha384 * sha384,
    word32 * flags
)
```

Gets SHA384 flags.

Parameters:

- **sha384** SHA384 structure
- **flags** Pointer to store flags

See: wc_Sha384SetFlags

Return:

- 0 on success
- negative on error

Example

```
wc_Sha384 sha;
word32 flags;
int ret = wc_Sha384GetFlags(&sha, &flags);
```

C.53.2.50 function wc_Sha384Transform

```
int wc_Sha384Transform(
    wc_Sha384 * sha,
    const unsigned char * data
)
```

Transforms SHA384 block.

Parameters:

- **sha** SHA384 structure
- **data** Block data

See: wc_Sha384Update

Return:

- 0 on success
- negative on error

Example

```
wc_Sha384 sha;
unsigned char block[WC_SHA384_BLOCK_SIZE];
int ret = wc_Sha384Transform(&sha, block);
```

C.53.3 Source code

```
int wc_InitSha512(wc_Sha512* sha);

int wc_Sha512Update(wc_Sha512* sha, const byte* data, word32 len);

int wc_Sha512Final(wc_Sha512* sha512, byte* hash);
```

```
int wc_InitSha384(wc_Sha384* sha);

int wc_Sha384Update(wc_Sha384* sha, const byte* data, word32 len);

int wc_Sha384Final(wc_Sha384* sha384, byte* hash);

int wc_InitSha512_ex(wc_Sha512* sha, void* heap, int devId);

int wc_Sha512FinalRaw(wc_Sha512* sha512, byte* hash);

void wc_Sha512Free(wc_Sha512* sha);

        wc_Sha512Update(&sha, data2, sizeof(data2));
        wc_Sha512Final(&sha, hash);
        wc_Sha512Free(&sha);
    }
    \endcode

    \sa wc_InitSha512
    \sa wc_Sha512Final
    \sa wc_Sha512Free
*/
int wc_Sha512Reset(wc_Sha512* sha);

int wc_Sha512GetHash(wc_Sha512* sha512, byte* hash);

int wc_Sha512Copy(wc_Sha512* src, wc_Sha512* dst);

int wc_Sha512_Grow(wc_Sha512* sha512, const byte* in, int inSz);

int wc_Sha512SetFlags(wc_Sha512* sha512, word32 flags);

int wc_Sha512GetFlags(wc_Sha512* sha512, word32* flags);

int wc_Sha512Transform(wc_Sha512* sha, const unsigned char* data);

int wc_InitSha512_224(wc_Sha512* sha);

int wc_InitSha512_224_ex(wc_Sha512* sha, void* heap, int devId);

int wc_Sha512_224Update(wc_Sha512* sha, const byte* data, word32 len);

int wc_Sha512_224FinalRaw(wc_Sha512* sha512, byte* hash);

int wc_Sha512_224Final(wc_Sha512* sha512, byte* hash);

void wc_Sha512_224Free(wc_Sha512* sha);

        wc_Sha512_224Update(&sha, data2, sizeof(data2));
        wc_Sha512_224Final(&sha, hash);
        wc_Sha512_224Free(&sha);
    }
    \endcode
```

```
    \sa wc_InitSha512_224
    \sa wc_Sha512_224Final
    \sa wc_Sha512_224Free
*/
int wc_Sha512_224Reset(wc_Sha512* sha);

int wc_Sha512_224GetHash(wc_Sha512* sha512, byte* hash);

int wc_Sha512_224Copy(wc_Sha512* src, wc_Sha512* dst);

int wc_Sha512_224SetFlags(wc_Sha512* sha512, word32 flags);

int wc_Sha512_224GetFlags(wc_Sha512* sha512, word32* flags);

int wc_Sha512_224Transform(wc_Sha512* sha, const unsigned char* data);

int wc_InitSha512_256(wc_Sha512* sha);

int wc_InitSha512_256_ex(wc_Sha512* sha, void* heap, int devId);

int wc_Sha512_256Update(wc_Sha512* sha, const byte* data, word32 len);

int wc_Sha512_256FinalRaw(wc_Sha512* sha512, byte* hash);

int wc_Sha512_256Final(wc_Sha512* sha512, byte* hash);

void wc_Sha512_256Free(wc_Sha512* sha);

        wc_Sha512_256Update(&sha, data2, sizeof(data2));
        wc_Sha512_256Final(&sha, hash);
        wc_Sha512_256Free(&sha);
    }
    \endcode

    \sa wc_InitSha512_256
    \sa wc_Sha512_256Final
    \sa wc_Sha512_256Free
*/
int wc_Sha512_256Reset(wc_Sha512* sha);

int wc_Sha512_256GetHash(wc_Sha512* sha512, byte* hash);

int wc_Sha512_256Copy(wc_Sha512* src, wc_Sha512* dst);

int wc_Sha512_256SetFlags(wc_Sha512* sha512, word32 flags);

int wc_Sha512_256GetFlags(wc_Sha512* sha512, word32* flags);

int wc_Sha512_256Transform(wc_Sha512* sha, const unsigned char* data);

int wc_InitSha384_ex(wc_Sha384* sha, void* heap, int devId);

int wc_Sha384FinalRaw(wc_Sha384* sha384, byte* hash);
```

```
void wc_Sha384Free(wc_Sha384* sha);

        wc_Sha384Update(&sha, data2, sizeof(data2));
        wc_Sha384Final(&sha, hash);
        wc_Sha384Free(&sha);
    }
    \endcode

    \sa wc_InitSha384
    \sa wc_Sha384Final
    \sa wc_Sha384Free
*/
int wc_Sha384Reset(wc_Sha384* sha);

int wc_Sha384GetHash(wc_Sha384* sha384, byte* hash);

int wc_Sha384Copy(wc_Sha384* src, wc_Sha384* dst);

int wc_Sha384_Grow(wc_Sha384* sha384, const byte* in, int inSz);

int wc_Sha384SetFlags(wc_Sha384* sha384, word32 flags);

int wc_Sha384GetFlags(wc_Sha384* sha384, word32* flags);

int wc_Sha384Transform(wc_Sha384* sha, const unsigned char* data);
```

C.54 sha.h

C.54.1 Functions

	Name
int	**wc_InitSha**(wc_Sha * sha)This function initializes SHA. This is automatically called by wc_ShaHash.
int	**wc_ShaUpdate**(wc_Sha * sha, const byte * data, word32 len)Can be called to continually hash the provided byte array of length len.
int	**wc_ShaFinal**(wc_Sha * sha, byte * hash)Finalizes hashing of data. Result is placed into hash. Resets state of sha struct.
void	**wc_ShaFree**(wc_Sha * sha)Used to clean up memory used by an initialized Sha struct.
int	**wc_ShaGetHash**(wc_Sha * sha, byte * hash)Gets hash data. Result is placed into hash. Does not reset state of sha struct.
int	**wc_InitSha_ex**(wc_Sha * sha, void * heap, int devId)Initializes SHA with heap and device ID.
int	**wc_ShaFinalRaw**(wc_Sha * sha, byte * hash)Gets raw hash without finalizing.
int	**wc_ShaCopy**(wc_Sha * src, wc_Sha * dst)Copies SHA context.
int	**wc_ShaTransform**(wc_Sha * sha, const unsigned char * data)Transforms SHA block.

	Name
void	**wc_ShaSizeSet**(wc_Sha * sha, word32 len)Sets SHA size.
int	**wc_ShaSetFlags**(wc_Sha * sha, word32 flags)Sets SHA flags.

C.54.2 Functions Documentation

C.54.2.1 function wc_InitSha

```
int wc_InitSha(
    wc_Sha * sha
)
```

This function initializes SHA. This is automatically called by wc_ShaHash.

Parameters:

- **sha** pointer to the sha structure to use for encryption

See:

- wc_ShaHash
- wc_ShaUpdate
- wc_ShaFinal

Return: 0 Returned upon successfully initializing

Example

```
Sha sha[1];
if ((ret = wc_InitSha(sha)) != 0) {
   WOLFSSL_MSG("wc_InitSha failed");
}
else {
   wc_ShaUpdate(sha, data, len);
   wc_ShaFinal(sha, hash);
}
```

C.54.2.2 function wc_ShaUpdate

```
int wc_ShaUpdate(
    wc_Sha * sha,
    const byte * data,
    word32 len
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **sha** pointer to the sha structure to use for encryption
- **data** the data to be hashed
- **len** length of data to be hashed

See:

- wc_ShaHash
- wc_ShaFinal
- wc_InitSha

Return: 0 Returned upon successfully adding the data to the digest.

Example

```
Sha sha[1];
byte data[] = { // Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha(sha)) != 0) {
   WOLFSSL_MSG("wc_InitSha failed");
}
else {
   wc_ShaUpdate(sha, data, len);
   wc_ShaFinal(sha, hash);
}
```

C.54.2.3 function wc_ShaFinal

```
int wc_ShaFinal(
    wc_Sha * sha,
    byte * hash
)
```

Finalizes hashing of data. Result is placed into hash. Resets state of sha struct.

Parameters:

- **sha** pointer to the sha structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_ShaHash
- wc_InitSha
- wc_ShaGetHash

Return: 0 Returned upon successfully finalizing.

Example

```
Sha sha[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha(sha)) != 0) {
   WOLFSSL_MSG("wc_InitSha failed");
}
else {
   wc_ShaUpdate(sha, data, len);
   wc_ShaFinal(sha, hash);
}
```

C.54.2.4 function wc_ShaFree

```
void wc_ShaFree(
    wc_Sha * sha
)
```

Used to clean up memory used by an initialized Sha struct.

Parameters:

- **sha** Pointer to the Sha struct to free.

See:

- wc_InitSha
- wc_ShaUpdate
- wc_ShaFinal

Return: No returns.

Example

```
Sha sha;
wc_InitSha(&sha);
// Use sha
wc_ShaFree(&sha);
```

C.54.2.5 function wc_ShaGetHash

```
int wc_ShaGetHash(
    wc_Sha * sha,
    byte * hash
)
```

Gets hash data. Result is placed into hash. Does not reset state of sha struct.

Parameters:

- **sha** pointer to the sha structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_ShaHash
- wc_ShaFinal
- wc_InitSha

Return: 0 Returned upon successfully finalizing.

Example

```
Sha sha[1];
if ((ret = wc_InitSha(sha)) != 0) {
WOLFSSL_MSG("wc_InitSha failed");
}
else {
    wc_ShaUpdate(sha, data, len);
    wc_ShaGetHash(sha, hash);
}
```

C.54.2.6 function wc_InitSha_ex

```
int wc_InitSha_ex(
    wc_Sha * sha,
    void * heap,
    int devId
)
```

Initializes SHA with heap and device ID.

Parameters:

- **sha** SHA structure
- **heap** Heap hint
- **devId** Device ID

See: wc_InitSha

Return:

- 0 on success
- negative on error

Example

```
wc_Sha sha;
int ret = wc_InitSha_ex(&sha, NULL, INVALID_DEVID);
```

C.54.2.7 function wc_ShaFinalRaw

```
int wc_ShaFinalRaw(
    wc_Sha * sha,
    byte * hash
)
```

Gets raw hash without finalizing.

Parameters:

- **sha** SHA structure
- **hash** Output hash buffer

See: wc_ShaFinal

Return:

- 0 on success
- negative on error

Example

```
wc_Sha sha;
byte hash[WC_SHA_DIGEST_SIZE];
int ret = wc_ShaFinalRaw(&sha, hash);
```

C.54.2.8 function wc_ShaCopy

```
int wc_ShaCopy(
    wc_Sha * src,
    wc_Sha * dst
)
```

Copies SHA context.

Parameters:

- **src** Source SHA structure
- **dst** Destination SHA structure (must be zeroed or previously initialized)

See: wc_InitSha

Return:

- 0 on success
- negative on error

Example

```
wc_Sha src, dst;
memset(&dst, 0, sizeof(dst));
int ret = wc_InitSha(&src);
if (ret == 0)
    ret = wc_ShaCopy(&src, &dst);
```

C.54.2.9 function wc_ShaTransform

```
int wc_ShaTransform(
    wc_Sha * sha,
    const unsigned char * data
)
```

Transforms SHA block.

Parameters:

- **sha** SHA structure
- **data** Block data

See: wc_ShaUpdate

Return:

- 0 on success
- negative on error

Example

```
wc_Sha sha;
unsigned char block[WC_SHA_BLOCK_SIZE];
int ret = wc_ShaTransform(&sha, block);
```

C.54.2.10 function wc_ShaSizeSet

```
void wc_ShaSizeSet(
    wc_Sha * sha,
    word32 len
)
```

Sets SHA size.

Parameters:

- **sha** SHA structure
- **len** Size to set

See: wc_ShaUpdate

Return: none No returns

Example

```
wc_Sha sha;
wc_ShaSizeSet(&sha, 1000);
```

C.54.2.11 function wc_ShaSetFlags

```
int wc_ShaSetFlags(
    wc_Sha * sha,
    word32 flags
)
```

Sets SHA flags.

Parameters:

- **sha** SHA structure
- **flags** Flags to set

See: wc_InitSha

Return:

- 0 on success
- negative on error

Example

```
wc_Sha sha;
int ret = wc_ShaSetFlags(&sha, WC_HASH_FLAG_WILLCOPY);
```

C.54.3 Source code

```
int wc_InitSha(wc_Sha* sha);

int wc_ShaUpdate(wc_Sha* sha, const byte* data, word32 len);

int wc_ShaFinal(wc_Sha* sha, byte* hash);

void wc_ShaFree(wc_Sha* sha);

int wc_ShaGetHash(wc_Sha* sha, byte* hash);
int wc_InitSha_ex(wc_Sha* sha, void* heap, int devId);

int wc_ShaFinalRaw(wc_Sha* sha, byte* hash);

int wc_ShaCopy(wc_Sha* src, wc_Sha* dst);

int wc_ShaTransform(wc_Sha* sha, const unsigned char* data);

void wc_ShaSizeSet(wc_Sha* sha, word32 len);

int wc_ShaSetFlags(wc_Sha* sha, word32 flags);
```

C.55 signature.h

C.55.1 Functions

	Name
int	**wc_SignatureGetSize**(enum wc_SignatureType sig_type, const void * key, word32 key_len)This function returns the maximum size of the resulting signature.
int	**wc_SignatureVerify**(enum wc_HashType hash_type, enum wc_SignatureType sig_type, const byte * data, word32 data_len, const byte * sig, word32 sig_len, void * key, word32 key_len)This function validates a signature by hashing the data and using the resulting hash and key to verify the signature.
int	**wc_SignatureGenerate**(enum wc_HashType hash_type, enum wc_SignatureType sig_type, const byte * data, word32 data_len, byte * sig, word32 * sig_len, void * key, word32 key_len, WC_RNG * rng)This function generates a signature from the data using a key. It first creates a hash of the data then signs the hash using the key.
int	**wc_SignatureVerifyHash**(enum wc_HashType hash_type, enum wc_SignatureType sig_type, const byte * hash_data, word32 hash_len, const byte * sig, word32 sig_len, void * key, word32 key_len)This function verifies a signature using a pre-computed hash. Unlike wc_SignatureVerify which hashes the data first, this function takes the hash directly and verifies the signature against it. If sig_type is WC_SIGNATURE_TYPE_RSA_W_ENC, hash data must be encoded with wc_EncodeSignature prior to calling.
int	**wc_SignatureGenerateHash**(enum wc_HashType hash_type, enum wc_SignatureType sig_type, const byte * hash_data, word32 hash_len, byte * sig, word32 * sig_len, void * key, word32 key_len, WC_RNG * rng)This function generates a signature from a pre-computed hash. Unlike wc_SignatureGenerate which hashes the data first, this function takes the hash directly and signs it. If sig_type is WC_SIGNATURE_TYPE_RSA_W_ENC, hash data must be encoded with wc_EncodeSignature prior to calling.

	Name
int	**wc_SignatureGenerateHash_ex**(enum wc_HashType hash_type, enum wc_SignatureType sig_type, const byte * hash_data, word32 hash_len, byte * sig, word32 * sig_len, void * key, word32 key_len, WC_RNG * rng, int verify)This function generates a signature from a pre-computed hash with extended options. This is similar to wc_SignatureGenerateHash but allows optional verification of the signature after generation.
int	**wc_SignatureGenerate_ex**(enum wc_HashType hash_type, enum wc_SignatureType sig_type, const byte * data, word32 data_len, byte * sig, word32 * sig_len, void * key, word32 key_len, WC_RNG * rng, int verify)This function generates a signature from data with extended options. This is similar to wc_SignatureGenerate but allows optional verification of the signature after generation.

C.55.2 Functions Documentation

C.55.2.1 function wc_SignatureGetSize

```
int wc_SignatureGetSize(
    enum wc_SignatureType sig_type,
    const void * key,
    word32 key_len
)
```

This function returns the maximum size of the resulting signature.

Parameters:

- **sig_type** A signature type enum value such as WC_SIGNATURE_TYPE_ECC or WC_SIGNATURE_TYPE_RSA.
- **key** Pointer to the key structure corresponding to sig_type: pass an ecc_key* for WC_SIGNATURE_TYPE_ECC, or a RsaKey* for WC_SIGNATURE_TYPE_RSA / WC_SIGNATURE_TYPE_RSA_W_ENC. The caller is responsible for ensuring the pointer refers to the correct type; this function cannot verify the actual runtime type of the object.
- **key_len** If key is non-NULL, key_len must be exactly sizeof(ecc_key) or sizeof(RsaKey) matching the sig_type. Passing any other value causes the function to return BAD_FUNC_ARG without dereferencing key. Always pass the size of the concrete key type at the call site: if you have a typed pointer (e.g., ecc_key* k), use sizeof(k); otherwise use sizeof(ecc_key) or sizeof(RsaKey) directly. Do not use sizeof(*key) on the const void parameter itself, as dereferencing void is invalid.

See:

- wc_HashGetDigestSize
- wc_SignatureGenerate
- wc_SignatureVerify

Return: Returns SIG_TYPE_E if sig_type is not supported. Returns BAD_FUNC_ARG if sig_type was invalid or key_len does not exactly match the size of the expected key structure. A positive return value indicates the maximum size of a signature.

Example

```
// Get signature length
enum wc_SignatureType sig_type = WC_SIGNATURE_TYPE_ECC;
ecc_key eccKey;
word32 sigLen;
wc_ecc_init(&eccKey);
sigLen = wc_SignatureGetSize(sig_type, &eccKey, sizeof(eccKey));
if (sigLen > 0) {
    // Success
}
```

C.55.2.2 function wc_SignatureVerify

```
int wc_SignatureVerify(
    enum wc_HashType hash_type,
    enum wc_SignatureType sig_type,
    const byte * data,
    word32 data_len,
    const byte * sig,
    word32 sig_len,
    void * key,
    word32 key_len
)
```

This function validates a signature by hashing the data and using the resulting hash and key to verify the signature.

Parameters:

- **hash_type** A hash type from the “enum wc_HashType” such as “WC_HASH_TYPE_SHA256”.
- **sig_type** A signature type enum value such as WC_SIGNATURE_TYPE_ECC or WC_SIGNATURE_TYPE_RSA.
- **data** Pointer to buffer containing the data to hash.
- **data_len** Length of the data buffer.
- **sig** Pointer to buffer to output signature.
- **sig_len** Length of the signature output buffer.
- **key** Pointer to the key structure corresponding to sig_type. See wc_SignatureGetSize() for the type-safety constraints that apply to this parameter.
- **key_len** Must be exactly sizeof(ecc_key) or sizeof(RsaKey) matching sig_type. See wc_SignatureGetSize().

See:

- wc_SignatureGetSize
- wc_SignatureGenerate

Return:

- 0 Success
- SIG_TYPE_E -231, signature type not enabled/ available
- BAD_FUNC_ARG -173, bad function argument provided
- BUFFER_E -132, output buffer too small or input too large.

Example

```
int ret;
ecc_key eccKey;

// Import the public key
wc_ecc_init(&eccKey);
ret = wc_ecc_import_x963(eccPubKeyBuf, eccPubKeyLen, &eccKey);
// Perform signature verification using public key
```

```
ret = wc_SignatureVerify(
WC_HASH_TYPE_SHA256, WC_SIGNATURE_TYPE_ECC,
fileBuf, fileLen,
sigBuf, sigLen,
&eccKey, sizeof(eccKey));
printf("Signature Verification: %s
(%d)\n", (ret == 0) ? "Pass" : "Fail", ret);
wc_ecc_free(&eccKey);
```

C.55.2.3 function wc_SignatureGenerate

```
int wc_SignatureGenerate(
    enum wc_HashType hash_type,
    enum wc_SignatureType sig_type,
    const byte * data,
    word32 data_len,
    byte * sig,
    word32 * sig_len,
    void * key,
    word32 key_len,
    WC_RNG * rng
)
```

This function generates a signature from the data using a key. It first creates a hash of the data then signs the hash using the key.

Parameters:

- **hash_type** A hash type from the "enum wc_HashType" such as "WC_HASH_TYPE_SHA256".
- **sig_type** A signature type enum value such as WC_SIGNATURE_TYPE_ECC or WC_SIGNATURE_TYPE_RSA.
- **data** Pointer to buffer containing the data to hash.
- **data_len** Length of the data buffer.
- **sig** Pointer to buffer to output signature.
- **sig_len** Length of the signature output buffer.
- **key** Pointer to the key structure corresponding to sig_type. See wc_SignatureGetSize() for the type-safety constraints that apply to this parameter.
- **key_len** Must be exactly sizeof(ecc_key) or sizeof(RsaKey) matching sig_type. See wc_SignatureGetSize().
- **rng** Pointer to an initialized RNG structure.

See:

- wc_SignatureGetSize
- wc_SignatureVerify

Return:

- 0 Success
- SIG_TYPE_E -231, signature type not enabled/ available
- BAD_FUNC_ARG -173, bad function argument provided
- BUFFER_E -132, output buffer too small or input too large.

Example

```
int ret;
WC_RNG rng;
ecc_key eccKey;

wc_InitRng(&rng);
wc_ecc_init(&eccKey);
```

```
// Generate key
ret = wc_ecc_make_key(&rng, 32, &eccKey);

// Get signature length and allocate buffer
sigLen = wc_SignatureGetSize(sig_type, &eccKey, sizeof(eccKey));
sigBuf = malloc(sigLen);

// Perform signature verification using public key
ret = wc_SignatureGenerate(
    WC_HASH_TYPE_SHA256, WC_SIGNATURE_TYPE_ECC,
    fileBuf, fileLen,
    sigBuf, &sigLen,
    &eccKey, sizeof(eccKey),
    &rng);
printf("Signature Generation: %s
(%d)\n", (ret == 0) ? "Pass" : "Fail", ret);

free(sigBuf);
wc_ecc_free(&eccKey);
wc_FreeRng(&rng);
```

C.55.2.4 function wc_SignatureVerifyHash

```
int wc_SignatureVerifyHash(
    enum wc_HashType hash_type,
    enum wc_SignatureType sig_type,
    const byte * hash_data,
    word32 hash_len,
    const byte * sig,
    word32 sig_len,
    void * key,
    word32 key_len
)
```

This function verifies a signature using a pre-computed hash. Unlike wc_SignatureVerify which hashes the data first, this function takes the hash directly and verifies the signature against it. If sig_type is WC_SIGNATURE_TYPE_RSA_W_ENC, hash data must be encoded with wc_EncodeSignature prior to calling.

Parameters:

- **hash_type** A hash type from enum wc_HashType
- **sig_type** A signature type such as WC_SIGNATURE_TYPE_ECC or WC_SIGNATURE_TYPE_RSA
- **hash_data** Pointer to buffer containing the hash to verify
- **hash_len** Length of the hash buffer
- **sig** Pointer to buffer containing the signature
- **sig_len** Length of the signature buffer
- **key** Pointer to the key structure corresponding to sig_type. See wc_SignatureGetSize() for the type-safety constraints that apply to this parameter.
- **key_len** Must be exactly sizeof(ecc_key) or sizeof(RsaKey) matching sig_type. See wc_SignatureGetSize().

See:

- wc_SignatureVerify
- wc_SignatureGenerateHash

Return:

- 0 Success
- SIG_TYPE_E Signature type not enabled/available
- BAD_FUNC_ARG Bad function argument provided
- BUFFER_E Output buffer too small or input too large

Example

```
ecc_key eccKey;
byte hash[WC_SHA256_DIGEST_SIZE];
byte sig[ECC_MAX_SIG_SIZE];
word32 sigLen = sizeof(sig);

wc_ecc_init(&eccKey);
// import public key, signature, and pre-computed hash ...
int ret = wc_SignatureVerifyHash(WC_HASH_TYPE_SHA256,
                                 WC_SIGNATURE_TYPE_ECC, hash,
                                 sizeof(hash), sig, sigLen,
                                 &eccKey, sizeof(eccKey));
if (ret == 0) {
    // signature verified
}
```

C.55.2.5 function wc_SignatureGenerateHash

```
int wc_SignatureGenerateHash(
    enum wc_HashType hash_type,
    enum wc_SignatureType sig_type,
    const byte * hash_data,
    word32 hash_len,
    byte * sig,
    word32 * sig_len,
    void * key,
    word32 key_len,
    WC_RNG * rng
)
```

This function generates a signature from a pre-computed hash. Unlike wc_SignatureGenerate which hashes the data first, this function takes the hash directly and signs it. If sig_type is WC_SIGNATURE_TYPE_RSA_W_ENC, hash data must be encoded with wc_EncodeSignature prior to calling.

Parameters:

- **hash_type** A hash type from enum wc_HashType
- **sig_type** A signature type such as WC_SIGNATURE_TYPE_ECC or WC_SIGNATURE_TYPE_RSA
- **hash_data** Pointer to buffer containing the hash to sign
- **hash_len** Length of the hash buffer
- **sig** Pointer to buffer to output signature
- **sig_len** Pointer to length of signature output buffer
- **key** Pointer to the key structure corresponding to sig_type. See wc_SignatureGetSize() for the type-safety constraints that apply to this parameter.
- **key_len** Must be exactly sizeof(ecc_key) or sizeof(RsaKey) matching sig_type. See wc_SignatureGetSize().
- **rng** Pointer to an initialized RNG structure

See:

- wc_SignatureGenerate
- wc_SignatureVerifyHash

Return:

- 0 Success
- SIG_TYPE_E Signature type not enabled/available
- BAD_FUNC_ARG Bad function argument provided
- BUFFER_E Output buffer too small or input too large

Example

```
WC_RNG rng;
ecc_key eccKey;
byte hash[WC_SHA256_DIGEST_SIZE];
byte sig[ECC_MAX_SIG_SIZE];
word32 sigLen = sizeof(sig);

wc_InitRng(&rng);
wc_ecc_init(&eccKey);
wc_ecc_make_key(&rng, 32, &eccKey);
// generate signature from pre-computed hash
int ret = wc_SignatureGenerateHash(WC_HASH_TYPE_SHA256,
                                   WC_SIGNATURE_TYPE_ECC, hash,
                                   sizeof(hash), sig, &sigLen,
                                   &eccKey, sizeof(eccKey), &rng);
```

C.55.2.6 function wc_SignatureGenerateHash_ex

```
int wc_SignatureGenerateHash_ex(
    enum wc_HashType hash_type,
    enum wc_SignatureType sig_type,
    const byte * hash_data,
    word32 hash_len,
    byte * sig,
    word32 * sig_len,
    void * key,
    word32 key_len,
    WC_RNG * rng,
    int verify
)
```

This function generates a signature from a pre-computed hash with extended options. This is similar to wc_SignatureGenerateHash but allows optional verification of the signature after generation.

Parameters:

- **hash_type** A hash type from enum wc_HashType
- **sig_type** A signature type such as WC_SIGNATURE_TYPE_ECC or WC_SIGNATURE_TYPE_RSA
- **hash_data** Pointer to buffer containing the hash to sign
- **hash_len** Length of the hash buffer
- **sig** Pointer to buffer to output signature
- **sig_len** Pointer to length of signature output buffer
- **key** Pointer to the key structure corresponding to sig_type. See wc_SignatureGetSize() for the type-safety constraints that apply to this parameter.
- **key_len** Must be exactly sizeof(ecc_key) or sizeof(RsaKey) matching sig_type. See wc_SignatureGetSize().
- **rng** Pointer to an initialized RNG structure
- **verify** If non-zero, verify the signature after generation

See:

- wc_SignatureGenerateHash
- wc_SignatureGenerate_ex

Return:

- 0 Success
- SIG_TYPE_E Signature type not enabled/available
- BAD_FUNC_ARG Bad function argument provided
- BUFFER_E Output buffer too small or input too large

Example

```
WC_RNG rng;
ecc_key eccKey;
byte hash[WC_SHA256_DIGEST_SIZE];
byte sig[ECC_MAX_SIG_SIZE];
word32 sigLen = sizeof(sig);

wc_InitRng(&rng);
wc_ecc_init(&eccKey);
wc_ecc_make_key(&rng, 32, &eccKey);
int ret = wc_SignatureGenerateHash_ex(WC_HASH_TYPE_SHA256,
                                      WC_SIGNATURE_TYPE_ECC, hash,
                                      sizeof(hash), sig, &sigLen,
                                      &eccKey, sizeof(eccKey),
                                      &rng, 1);
```

C.55.2.7 function wc_SignatureGenerate_ex

```
int wc_SignatureGenerate_ex(
    enum wc_HashType hash_type,
    enum wc_SignatureType sig_type,
    const byte * data,
    word32 data_len,
    byte * sig,
    word32 * sig_len,
    void * key,
    word32 key_len,
    WC_RNG * rng,
    int verify
)
```

This function generates a signature from data with extended options. This is similar to wc_SignatureGenerate but allows optional verification of the signature after generation.

Parameters:

- **hash_type** A hash type from enum wc_HashType
- **sig_type** A signature type such as WC_SIGNATURE_TYPE_ECC or WC_SIGNATURE_TYPE_RSA
- **data** Pointer to buffer containing the data to hash and sign
- **data_len** Length of the data buffer
- **sig** Pointer to buffer to output signature
- **sig_len** Pointer to length of signature output buffer
- **key** Pointer to the key structure corresponding to sig_type. See wc_SignatureGetSize() for the type-safety constraints that apply to this parameter.
- **key_len** Must be exactly sizeof(ecc_key) or sizeof(RsaKey) matching sig_type. See wc_SignatureGetSize().

- **rng** Pointer to an initialized RNG structure
- **verify** If non-zero, verify the signature after generation

See:

- wc_SignatureGenerate
- wc_SignatureGenerateHash_ex

Return:

- 0 Success
- SIG_TYPE_E Signature type not enabled/available
- BAD_FUNC_ARG Bad function argument provided
- BUFFER_E Output buffer too small or input too large

Example

```
WC_RNG rng;
ecc_key eccKey;
byte data[]; // data to sign
byte sig[ECC_MAX_SIG_SIZE];
word32 sigLen = sizeof(sig);

wc_InitRng(&rng);
wc_ecc_init(&eccKey);
wc_ecc_make_key(&rng, 32, &eccKey);
int ret = wc_SignatureGenerate_ex(WC_HASH_TYPE_SHA256,
                                  WC_SIGNATURE_TYPE_ECC, data,
                                  sizeof(data), sig, &sigLen,
                                  &eccKey, sizeof(eccKey), &rng, 1);
```

C.55.3 Source code

```
int wc_SignatureGetSize(enum wc_SignatureType sig_type,
    const void* key, word32 key_len);

int wc_SignatureVerify(
    enum wc_HashType hash_type, enum wc_SignatureType sig_type,
    const byte* data, word32 data_len,
    const byte* sig, word32 sig_len,
    void* key, word32 key_len);

int wc_SignatureGenerate(
    enum wc_HashType hash_type, enum wc_SignatureType sig_type,
    const byte* data, word32 data_len,
    byte* sig, word32 *sig_len,
    void* key, word32 key_len,
    WC_RNG* rng);

int wc_SignatureVerifyHash(enum wc_HashType hash_type,
                           enum wc_SignatureType sig_type,
                           const byte* hash_data, word32 hash_len,
                           const byte* sig, word32 sig_len,
                           void* key, word32 key_len);

int wc_SignatureGenerateHash(enum wc_HashType hash_type,
```

```
                                enum wc_SignatureType sig_type,
                                const byte* hash_data, word32 hash_len,
                                byte* sig, word32 *sig_len,
                                void* key, word32 key_len,
                                WC_RNG* rng);

int wc_SignatureGenerateHash_ex(enum wc_HashType hash_type,
                                   enum wc_SignatureType sig_type,
                                   const byte* hash_data, word32 hash_len,
                                   byte* sig, word32 *sig_len,
                                   void* key, word32 key_len,
                                   WC_RNG* rng, int verify);

int wc_SignatureGenerate_ex(enum wc_HashType hash_type,
                               enum wc_SignatureType sig_type,
                               const byte* data, word32 data_len,
                               byte* sig, word32 *sig_len,
                               void* key, word32 key_len,
                               WC_RNG* rng, int verify);
```

C.56 siphash.h

C.56.1 Functions

	Name
int	**wc_InitSipHash**(SipHash * siphash, const unsigned char * key, unsigned char outSz)This function initializes SipHash with a key for a MAC size.
int	**wc_SipHashUpdate**(SipHash * siphash, const unsigned char * in, word32 inSz)Can be called to continually hash the provided byte array of length len.
int	**wc_SipHashFinal**(SipHash * siphash, unsigned char * out, unsigned char outSz)Finalizes MACing of data. Result is placed into out.
int	**wc_SipHash**(const unsigned char * key, const unsigned char * in, word32 inSz, unsigned char * out, unsigned char outSz)This function one-shots the data using SipHash to calculate a MAC based on the key.

C.56.2 Functions Documentation

C.56.2.1 function wc_InitSipHash

```
int wc_InitSipHash(
    SipHash * siphash,
    const unsigned char * key,
    unsigned char outSz
)
```

This function initializes SipHash with a key for a MAC size.

Parameters:

- **siphash** pointer to the SipHash structure to use for MACing
- **key** pointer to the 16-byte array
- **outSz** number of bytes to output as MAC

See:

- wc_SipHash
- wc_SipHashUpdate
- wc_SipHashFinal

Return:

- 0 Returned upon successfully initializing
- BAD_FUNC_ARG Returned when siphash or key is NULL
- BAD_FUNC_ARG Returned when outSz is neither 8 nor 16

Example

```
SipHash siphash[1];
unsigned char key[16] = { ... };
byte macSz = 8; // 8 or 16

if ((ret = wc_InitSipHash(siphash, key, macSz)) != 0) {
    WOLFSSL_MSG("wc_InitSipHash failed");
}
else if ((ret = wc_SipHashUpdate(siphash, data, len)) != 0) {
    WOLFSSL_MSG("wc_SipHashUpdate failed");
}
else if ((ret = wc_SipHashFinal(siphash, mac, macSz)) != 0) {
    WOLFSSL_MSG("wc_SipHashFinal failed");
}
```

C.56.2.2 function wc_SipHashUpdate

```
int wc_SipHashUpdate(
    SipHash * siphash,
    const unsigned char * in,
    word32 inSz
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **siphash** pointer to the SipHash structure to use for MACing
- **in** the data to be MACed
- **inSz** size of data to be MACed

See:

- wc_SipHash
- wc_InitSipHash
- wc_SipHashFinal

Return:

- 0 Returned upon successfully adding the data to the MAC
- BAD_FUNC_ARG Returned when siphash is NULL
- BAD_FUNC_ARG Returned when in is NULL and inSz is not zero

Example

```
SipHash siphash[1];
byte data[] = { Data to be MACed };
word32 len = sizeof(data);

if ((ret = wc_InitSipHash(siphash, key, macSz)) != 0) {
    WOLFSSL_MSG("wc_InitSipHash failed");
}
else if ((ret = wc_SipHashUpdate(siphash, data, len)) != 0) {
    WOLFSSL_MSG("wc_SipHashUpdate failed");
}
else if ((ret = wc_SipHashFinal(siphash, mac, macSz)) != 0) {
    WOLFSSL_MSG("wc_SipHashFinal failed");
}
```

C.56.2.3 function wc_SipHashFinal

```
int wc_SipHashFinal(
    SipHash * siphash,
    unsigned char * out,
    unsigned char outSz
)
```

Finalizes MACing of data. Result is placed into out.

Parameters:

- **siphash** pointer to the SipHash structure to use for MACing
- **out** Byte array to hold MAC value
- **outSz** number of bytes to output as MAC

See:

- wc_SipHash
- wc_InitSipHash
- wc_SipHashUpdate

Return:

- 0 Returned upon successfully finalizing.
- BAD_FUNC_ARG Returned when siphash of out is NULL
- BAD_FUNC_ARG Returned when outSz is not the same as the initialized value

Example

```
SipHash siphash[1];
byte mac[8] = { ... }; // 8 or 16 bytes
byte macSz = sizeof(mac);

if ((ret = wc_InitSipHash(siphash, key, macSz)) != 0) {
    WOLFSSL_MSG("wc_InitSipHash failed");
}
else if ((ret = wc_SipHashUpdate(siphash, data, len)) != 0) {
    WOLFSSL_MSG("wc_SipHashUpdate failed");
}
else if ((ret = wc_SipHashFinal(siphash, mac, macSz)) != 0) {
    WOLFSSL_MSG("wc_SipHashFinal failed");
}
```

C.56.2.4 function wc_SipHash

```
int wc_SipHash(
    const unsigned char * key,
    const unsigned char * in,
    word32 inSz,
    unsigned char * out,
    unsigned char outSz
)
```

This function one-shots the data using SipHash to calculate a MAC based on the key.

Parameters:

- **key** pointer to the 16-byte array
- **in** the data to be MACed
- **inSz** size of data to be MACed
- **out** Byte array to hold MAC value
- **outSz** number of bytes to output as MAC

See:

- wc_InitSipHash
- wc_SipHashUpdate
- wc_SipHashFinal

Return:

- 0 Returned upon successfully MACing
- BAD_FUNC_ARG Returned when key or out is NULL
- BAD_FUNC_ARG Returned when in is NULL and inSz is not zero
- BAD_FUNC_ARG Returned when outSz is neither 8 nor 16

Example

```
unsigned char key[16] = { ... };
byte data[] = { Data to be MACed };
word32 len = sizeof(data);
byte mac[8] = { ... }; // 8 or 16 bytes
byte macSz = sizeof(mac);

if ((ret = wc_SipHash(key, data, len, mac, macSz)) != 0) {
    WOLFSSL_MSG("wc_SipHash failed");
}
```

C.56.3 Source code

```
int wc_InitSipHash(SipHash* siphash, const unsigned char* key,
    unsigned char outSz);

int wc_SipHashUpdate(SipHash* siphash, const unsigned char* in,
    word32 inSz);

int wc_SipHashFinal(SipHash* siphash, unsigned char* out,
    unsigned char outSz);

int wc_SipHash(const unsigned char* key, const unsigned char* in,
    word32 inSz, unsigned char* out, unsigned char outSz);
```

C.57 srp.h

C.57.1 Functions

	Name
int	**wc_SrpInit**(Srp * srp, SrpType type, SrpSide side)Initializes the Srp struct for usage.
int	**wc_SrpInit_ex**(Srp * srp, SrpType type, SrpSide side, void * heap, int devId)Initializes the Srp struct for usage with extended parameters. This function is similar to wc_SrpInit but allows specification of a custom heap hint and device ID for hardware acceleration.
void	**wc_SrpTerm**(Srp * srp)Releases the Srp struct resources after usage.
int	**wc_SrpSetUsername**(Srp * srp, const byte * username, word32 size)Sets the username. This function MUST be called after wc_SrpInit.
int	**wc_SrpSetParams**(Srp * srp, const byte * N, word32 nSz, const byte * g, word32 gSz, const byte * salt, word32 saltSz)Sets the srp parameters based on the username.. Must be called after wc_SrpSetUsername.
int	**wc_SrpSetPassword**(Srp * srp, const byte * password, word32 size)Sets the password. Setting the password does not persists the clear password data in the srp structure. The client calculates x = H(salt + H(user:pswd)) and stores it in the auth field. This function MUST be called after wc_SrpSetParams and is CLIENT SIDE ONLY.
int	**wc_SrpSetVerifier**(Srp * srp, const byte * verifier, word32 size)Sets the verifier. This function MUST be called after wc_SrpSetParams and is SERVER SIDE ONLY.
int	**wc_SrpGetVerifier**(Srp * srp, byte * verifier, word32 * size)Gets the verifier. The client calculates the verifier with v = g ^ x % N. This function MAY be called after wc_SrpSetPassword and is CLIENT SIDE ONLY.
int	**wc_SrpSetPrivate**(Srp * srp, const byte * priv, word32 size)Sets the private ephemeral value. The private ephemeral value is known as: a at the client side. a = random() b at the server side. b = random() This function is handy for unit test cases or if the developer wants to use an external random source to set the ephemeral value. This function MAY be called before wc_SrpGetPublic.

	Name
int	**wc_SrpGetPublic**(Srp * srp, byte * pub, word32 * size)Gets the public ephemeral value. The public ephemeral value is known as: A at the client side. A = g ^ a % N B at the server side. B = (k * v + (g ^ b % N)) % N This function MUST be called after wc_SrpSetPassword or wc_SrpSetVerifier. The function wc_SrpSetPrivate may be called before wc_SrpGetPublic.
int	**wc_SrpComputeKey**(Srp * srp, byte * clientPubKey, word32 clientPubKeySz, byte * serverPubKey, word32 serverPubKeySz)Computes the session key. The key can be accessed at srp->key after success.
int	**wc_SrpGetProof**(Srp * srp, byte * proof, word32 * size)Gets the proof. This function MUST be called after wc_SrpComputeKey.
int	**wc_SrpVerifyPeersProof**(Srp * srp, byte * proof, word32 size)Verifies the peers proof. This function MUST be called before wc_SrpGetSessionKey.

C.57.2 Functions Documentation

C.57.2.1 function wc_SrpInit

```
int wc_SrpInit(
    Srp * srp,
    SrpType type,
    SrpSide side
)
```

Initializes the Srp struct for usage.

Parameters:

- **srp** the Srp structure to be initialized.
- **type** the hash type to be used.
- **side** the side of the communication.

See:

- wc_SrpTerm
- wc_SrpSetUsername

Return:

- 0 on success.
- BAD_FUNC_ARG Returns when there's an issue with the arguments such as srp being null or SrpSide not being SRP_CLIENT_SIDE or SRP_SERVER_SIDE.
- NOT_COMPILED_IN Returns when a type is passed as an argument but hasn't been configured in the wolfCrypt build.
- <0 on error.

Example

```
Srp srp;
if (wc_SrpInit(&srp, SRP_TYPE_SHA, SRP_CLIENT_SIDE) != 0)
{
    // Initialization error
}
else
{
    wc_SrpTerm(&srp);
}
```

C.57.2.2 function wc_SrpInit_ex

```
int wc_SrpInit_ex(
    Srp * srp,
    SrpType type,
    SrpSide side,
    void * heap,
    int devId
)
```

Initializes the Srp struct for usage with extended parameters. This function is similar to wc_SrpInit but allows specification of a custom heap hint and device ID for hardware acceleration.

Parameters:

- **srp** the Srp structure to be initialized.
- **type** the hash type to be used.
- **side** the side of the communication.
- **heap** pointer to heap hint for memory allocation (can be NULL).
- **devId** device ID for hardware acceleration (use INVALID_DEVID for software only).

See:

- wc_SrpInit
- wc_SrpTerm
- wc_SrpSetUsername

Return:

- 0 on success.
- BAD_FUNC_ARG Returns when there's an issue with the arguments such as srp being null or SrpSide not being SRP_CLIENT_SIDE or SRP_SERVER_SIDE.
- NOT_COMPILED_IN Returns when a type is passed as an argument but hasn't been configured in the wolfCrypt build.
- <0 on error.

Example

```
Srp srp;
void* heap = NULL;
int devId = INVALID_DEVID;

if (wc_SrpInit_ex(&srp, SRP_TYPE_SHA, SRP_CLIENT_SIDE, heap,
                  devId) != 0) {
    // Initialization error
}
else {
    wc_SrpTerm(&srp);
}
```

C.57.2.3 function wc_SrpTerm

```
void wc_SrpTerm(
    Srp * srp
)
```

Releases the Srp struct resources after usage.

Parameters:

- **srp** Pointer to the Srp structure to be terminated.

See: wc_SrpInit

Return: none No returns.

Example

```
Srp srp;
wc_SrpInit(&srp, SRP_TYPE_SHA, SRP_CLIENT_SIDE);
// Use srp
wc_SrpTerm(&srp)
```

C.57.2.4 function wc_SrpSetUsername

```
int wc_SrpSetUsername(
    Srp * srp,
    const byte * username,
    word32 size
)
```

Sets the username. This function MUST be called after wc_SrpInit.

Parameters:

- **srp** the Srp structure.
- **username** the buffer containing the username.
- **size** the username size in bytes

See:

- wc_SrpInit
- wc_SrpSetParams
- wc_SrpTerm

Return:

- 0 Username set successfully.
- BAD_FUNC_ARG: Return if srp or username is null.
- MEMORY_E: Returns if there is an issue allocating memory for srp->user
- < 0: Error.

Example

```
Srp srp;
byte username[] = "user";
word32 usernameSize = 4;

wc_SrpInit(&srp, SRP_TYPE_SHA, SRP_CLIENT_SIDE);
if(wc_SrpSetUsername(&srp, username, usernameSize) != 0)
{
    // Error occurred setting username.
```

```
}
wc_SrpTerm(&srp);
```

C.57.2.5 function wc_SrpSetParams

```
int wc_SrpSetParams(
    Srp * srp,
    const byte * N,
    word32 nSz,
    const byte * g,
    word32 gSz,
    const byte * salt,
    word32 saltSz
)
```

Sets the srp parameters based on the username.. Must be called after wc_SrpSetUsername.

Parameters:

- **srp** the Srp structure.
- **N** the Modulus. N = 2q+1, [q, N] are primes.
- **nSz** the N size in bytes.
- **g** the Generator modulo N.
- **gSz** the g size in bytes
- **salt** a small random salt. Specific for each username.
- **saltSz** the salt size in bytes

See:

- wc_SrpInit
- wc_SrpSetUsername
- wc_SrpTerm

Return:

- 0 Success
- BAD_FUNC_ARG Returns if srp, N, g, or salt is null or if nSz < gSz.
- SRP_CALL_ORDER_E Returns if wc_SrpSetParams is called before wc_SrpSetUsername.
- <0 Error

Example

```
Srp srp;
byte username[] = "user";
word32 usernameSize = 4;

byte N[] = { }; // Contents of byte array N
byte g[] = { }; // Contents of byte array g
byte salt[] = { }; // Contents of byte array salt

wc_SrpInit(&srp, SRP_TYPE_SHA, SRP_CLIENT_SIDE);
wc_SrpSetUsername(&srp, username, usernameSize);

if(wc_SrpSetParams(&srp, N, sizeof(N), g, sizeof(g), salt,
sizeof(salt)) != 0)
{
    // Error setting params
}
wc_SrpTerm(&srp);
```

C.57.2.6 function wc_SrpSetPassword

```
int wc_SrpSetPassword(
    Srp * srp,
    const byte * password,
    word32 size
)
```

Sets the password. Setting the password does not persists the clear password data in the srp structure. The client calculates x = H(salt + H(user:pswd)) and stores it in the auth field. This function MUST be called after wc_SrpSetParams and is CLIENT SIDE ONLY.

Parameters:

- **srp** The Srp structure.
- **password** The buffer containing the password.
- **size** The size of the password in bytes.

See:

- wc_SrpInit
- wc_SrpSetUsername
- wc_SrpSetParams

Return:

- 0 Success
- BAD_FUNC_ARG Returns if srp or password is null or if srp->side is not set to SRP_CLIENT_SIDE.
- SRP_CALL_ORDER_E Returns when wc_SrpSetPassword is called out of order.
- <0 Error

Example

```
Srp srp;
byte username[] = "user";
word32 usernameSize = 4;
byte password[] = "password";
word32 passwordSize = 8;

byte N[] = { }; // Contents of byte array N
byte g[] = { }; // Contents of byte array g
byte salt[] = { }; // Contents of byte array salt

wc_SrpInit(&srp, SRP_TYPE_SHA, SRP_CLIENT_SIDE);
wc_SrpSetUsername(&srp, username, usernameSize);
wc_SrpSetParams(&srp, N, sizeof(N), g, sizeof(g), salt, sizeof(salt));

if(wc_SrpSetPassword(&srp, password, passwordSize) != 0)
{
    // Error setting password
}

wc_SrpTerm(&srp);
```

C.57.2.7 function wc_SrpSetVerifier

```
int wc_SrpSetVerifier(
    Srp * srp,
    const byte * verifier,
```

```
    word32 size
)
```

Sets the verifier. This function MUST be called after wc_SrpSetParams and is SERVER SIDE ONLY.

Parameters:

- **srp** The Srp structure.
- **verifier** The structure containing the verifier.
- **size** The verifier size in bytes.

See:

- wc_SrpInit
- wc_SrpSetParams
- wc_SrpGetVerifier

Return:

- 0 Success
- BAD_FUNC_ARG Returned if srp or verifier is null or srp->side is not SRP_SERVER_SIDE.
- <0 Error

Example

```
Srp srp;
byte username[] = "user";
word32 usernameSize = 4;

byte N[] = { }; // Contents of byte array N
byte g[] = { }; // Contents of byte array g
byte salt[] = { }; // Contents of byte array salt
wc_SrpInit(&srp, SRP_TYPE_SHA, SRP_SERVER_SIDE);
wc_SrpSetUsername(&srp, username, usernameSize);
wc_SrpSetParams(&srp, N, sizeof(N), g, sizeof(g), salt, sizeof(salt))
byte verifier[] = { }; // Contents of some verifier

if(wc_SrpSetVerifier(&srp, verifier, sizeof(verifier)) != 0)
{
    // Error setting verifier
}

wc_SrpTerm(&srp);
```

C.57.2.8 function wc_SrpGetVerifier

```
int wc_SrpGetVerifier(
    Srp * srp,
    byte * verifier,
    word32 * size
)
```

Gets the verifier. The client calculates the verifier with v = g ^ x % N. This function MAY be called after wc_SrpSetPassword and is CLIENT SIDE ONLY.

Parameters:

- **srp** The Srp structure.
- **verifier** The buffer to write the verifier.
- **size** Buffer size in bytes. Updated with the verifier size.

See:

- wc_SrpSetVerifier
- wc_SrpSetPassword

Return:

- 0 Success
- BAD_FUNC_ARG Returned if srp, verifier or size is null or if srp->side is not SRP_CLIENT_SIDE.
- SRP_CALL_ORDER_E Returned if wc_SrpGetVerifier is called out of order.
- <0 Error

Example

```
Srp srp;
byte username[] = "user";
word32 usernameSize = 4;
byte password[] = "password";
word32 passwordSize = 8;

byte N[] = { }; // Contents of byte array N
byte g[] = { }; // Contents of byte array g
byte salt[] = { }; // Contents of byte array salt
byte v[64];
word32 vSz = 0;
vSz = sizeof(v);

wc_SrpInit(&srp, SRP_TYPE_SHA, SRP_CLIENT_SIDE);
wc_SrpSetUsername(&srp, username, usernameSize);
wc_SrpSetParams(&srp, N, sizeof(N), g, sizeof(g), salt, sizeof(salt))
wc_SrpSetPassword(&srp, password, passwordSize)

if( wc_SrpGetVerifier(&srp, v, &vSz ) != 0)
{
    // Error getting verifier
}
wc_SrpTerm(&srp);
```

C.57.2.9 function wc_SrpSetPrivate

```
int wc_SrpSetPrivate(
    Srp * srp,
    const byte * priv,
    word32 size
)
```

Sets the private ephemeral value. The private ephemeral value is known as: a at the client side. a = random() b at the server side. b = random() This function is handy for unit test cases or if the developer wants to use an external random source to set the ephemeral value. This function MAY be called before wc_SrpGetPublic.

Parameters:

- **srp** the Srp structure.
- **priv** the ephemeral value.
- **size** the private size in bytes.

See: wc_SrpGetPublic

Return:

- 0 Success
- BAD_FUNC_ARG Returned if srp, private, or size is null.
- SRP_CALL_ORDER_E Returned if wc_SrpSetPrivate is called out of order.
- <0 Error

Example

```
Srp srp;
byte username[] = "user";
word32 usernameSize = 4;

byte N[] = { }; // Contents of byte array N
byte g[] = { }; // Contents of byte array g
byte salt[] = { }; // Contents of byte array salt
byte verifier = { }; // Contents of some verifier
wc_SrpInit(&srp, SRP_TYPE_SHA, SRP_SERVER_SIDE);
wc_SrpSetUsername(&srp, username, usernameSize);
wc_SrpSetParams(&srp, N, sizeof(N), g, sizeof(g), salt, sizeof(salt))
wc_SrpSetVerifier(&srp, verifier, sizeof(verifier))

byte b[] = { }; // Some ephemeral value
if( wc_SrpSetPrivate(&srp, b, sizeof(b)) != 0)
{
    // Error setting private ephemeral
}

wc_SrpTerm(&srp);
```

C.57.2.10 function wc_SrpGetPublic

```
int wc_SrpGetPublic(
    Srp * srp,
    byte * pub,
    word32 * size
)
```

Gets the public ephemeral value. The public ephemeral value is known as: A at the client side. A = g ^ a % N B at the server side. B = (k * v + (g ^ b % N)) % N This function MUST be called after wc_SrpSetPassword or wc_SrpSetVerifier. The function wc_SrpSetPrivate may be called before wc_SrpGetPublic.

Parameters:

- **srp** the Srp structure.
- **pub** the buffer to write the public ephemeral value.
- **size** IN: the buffer size in bytes. OUT: Will be updated with the ephemeral value size.

See:

- wc_SrpSetPrivate
- wc_SrpSetPassword
- wc_SrpSetVerifier

Return:

- 0 Success
- BAD_FUNC_ARG Returned if srp, pub, or size is null.

- SRP_CALL_ORDER_E Returned if wc_SrpGetPublic is called out of order.
- BUFFER_E Returned if size < srp.N.
- <0 Error

Caller must observe value of size upon return to know the actual size.

Example

```
Srp srp;
byte username[] = "user";
word32 usernameSize = 4;
byte password[] = "password";
word32 passwordSize = 8;

byte N[] = { }; // Contents of byte array N
byte g[] = { }; // Contents of byte array g
byte salt[] = { }; // Contents of byte array salt
wc_SrpInit(&srp, SRP_TYPE_SHA, SRP_CLIENT_SIDE);
wc_SrpSetUsername(&srp, username, usernameSize);
wc_SrpSetParams(&srp, N, sizeof(N), g, sizeof(g), salt, sizeof(salt));
wc_SrpSetPassword(&srp, password, passwordSize)

byte public[64];
word32 publicSz = sizeof(public);

if( wc_SrpGetPublic(&srp, public, &publicSz) != 0)
{
    // Error getting public ephemeral
}

wc_SrpTerm(&srp);
```

C.57.2.11 function wc_SrpComputeKey

```
int wc_SrpComputeKey(
    Srp * srp,
    byte * clientPubKey,
    word32 clientPubKeySz,
    byte * serverPubKey,
    word32 serverPubKeySz
)
```

Computes the session key. The key can be accessed at srp->key after success.

Parameters:

- **srp** the Srp structure.
- **clientPubKey** the client's public ephemeral value.
- **clientPubKeySz** the client's public ephemeral value size.
- **serverPubKey** the server's public ephemeral value.
- **serverPubKeySz** the server's public ephemeral value size.

See: wc_SrpGetPublic

Return:

- 0 Success
- BAD_FUNC_ARG Returned if srp, clientPubKey, or serverPubKey or if clientPubKeySz or server-PubKeySz is 0.

- SRP_CALL_ORDER_E Returned if wc_SrpComputeKey is called out of order.
- <0 Error

Example

```
Srp server;

byte username[] = "user";
    word32 usernameSize = 4;
byte password[] = "password";
word32 passwordSize = 8;
byte N[] = { }; // Contents of byte array N
byte g[] = { }; // Contents of byte array g
byte salt[] = { }; // Contents of byte array salt
byte verifier[] = { }; // Contents of some verifier
byte serverPubKey[] = { }; // Contents of server pub key
word32 serverPubKeySize = sizeof(serverPubKey);
byte clientPubKey[64];
word32 clientPubKeySize = 64;

wc_SrpInit(&server, SRP_TYPE_SHA, SRP_SERVER_SIDE);
wc_SrpSetUsername(&server, username, usernameSize);
wc_SrpSetParams(&server, N, sizeof(N), g, sizeof(g), salt, sizeof(salt));
wc_SrpSetVerifier(&server, verifier, sizeof(verifier));
wc_SrpGetPublic(&server, serverPubKey, &serverPubKeySize);

wc_SrpComputeKey(&server, clientPubKey, clientPubKeySz,
                                    serverPubKey, serverPubKeySize)
wc_SrpTerm(&server);
```

C.57.2.12 function wc_SrpGetProof

```
int wc_SrpGetProof(
    Srp * srp,
    byte * proof,
    word32 * size
)
```

Gets the proof. This function MUST be called after wc_SrpComputeKey.

Parameters:

- **srp** the Srp structure.
- **proof** the peers proof.
- **size** the proof size in bytes.

See: wc_SrpComputeKey

Return:

- 0 Success
- BAD_FUNC_ARG Returns if srp, proof, or size is null.
- BUFFER_E Returns if size is less than the hash size of srp->type.
- <0 Error

Example

```
Srp cli;
byte clientProof[SRP_MAX_DIGEST_SIZE];
```

```
word32 clientProofSz = SRP_MAX_DIGEST_SIZE;

// Initialize Srp following steps from previous examples

if (wc_SrpGetProof(&cli, clientProof, &clientProofSz) != 0)
{
    // Error getting proof
}
```

C.57.2.13 function wc_SrpVerifyPeersProof

```
int wc_SrpVerifyPeersProof(
    Srp * srp,
    byte * proof,
    word32 size
)
```

Verifies the peers proof. This function MUST be called before wc_SrpGetSessionKey.

Parameters:

- **srp** the Srp structure.
- **proof** the peers proof.
- **size** the proof size in bytes.

See:

- wc_SrpGetSessionKey
- wc_SrpGetProof
- wc_SrpTerm

Return:

- 0 Success
- <0 Error

Example

```
Srp cli;
Srp srv;
byte clientProof[SRP_MAX_DIGEST_SIZE];
word32 clientProofSz = SRP_MAX_DIGEST_SIZE;

// Initialize Srp following steps from previous examples
// First get the proof
wc_SrpGetProof(&cli, clientProof, &clientProofSz)

if (wc_SrpVerifyPeersProof(&srv, clientProof, clientProofSz) != 0)
{
    // Error verifying proof
}
```

C.57.3 Source code

```
int wc_SrpInit(Srp* srp, SrpType type, SrpSide side);

int wc_SrpInit_ex(Srp* srp, SrpType type, SrpSide side, void* heap,
                  int devId);
```

```
void wc_SrpTerm(Srp* srp);

int wc_SrpSetUsername(Srp* srp, const byte* username, word32 size);

int wc_SrpSetParams(Srp* srp, const byte* N,    word32 nSz,
                                      const byte* g,    word32 gSz,
                                      const byte* salt, word32 saltSz);

int wc_SrpSetPassword(Srp* srp, const byte* password, word32 size);

int wc_SrpSetVerifier(Srp* srp, const byte* verifier, word32 size);

int wc_SrpGetVerifier(Srp* srp, byte* verifier, word32* size);

int wc_SrpSetPrivate(Srp* srp, const byte* priv, word32 size);

int wc_SrpGetPublic(Srp* srp, byte* pub, word32* size);

int wc_SrpComputeKey(Srp* srp,
                                 byte* clientPubKey, word32 clientPubKeySz,
                                 byte* serverPubKey, word32 serverPubKeySz);

int wc_SrpGetProof(Srp* srp, byte* proof, word32* size);

int wc_SrpVerifyPeersProof(Srp* srp, byte* proof, word32 size);
```

C.58 ssl.h

C.58.1 Types

	Name
typedef void()(*WOLFSSL* ssl, int client_server, int type, const byte *data, int len, void* arg)	**WOLFSSL_TLSEXT_DEBUG_CB** Callback type for the TLS extension debug callback.

C.58.2 Functions

	Name
WOLFSSL_METHOD *	**wolfDTLSv1_2_client_method_ex**(void * heap)This function initializes the DTLS v1.2 client method.
WOLFSSL_METHOD *	**wolfSSLv23_method**(void)This function returns a WOLFSSL_METHOD similar to wolfSSLv23_client_method except that it is not determined which side yet (server/client).

	Name
WOLFSSL_METHOD *	**wolfSSLv3_server_method**(void)The wolfSSLv3_server_method() function is used to indicate that the application is a server and will only support the SSL 3.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().
WOLFSSL_METHOD *	**wolfSSLv3_client_method**(void)The wolfSSLv3_client_method() function is used to indicate that the application is a client and will only support the SSL 3.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().
WOLFSSL_METHOD *	**wolfTLSv1_server_method**(void)The wolfTLSv1_server_method() function is used to indicate that the application is a server and will only support the TLS 1.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().
WOLFSSL_METHOD *	**wolfTLSv1_client_method**(void)The wolfTLSv1_client_method() function is used to indicate that the application is a client and will only support the TLS 1.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().
WOLFSSL_METHOD *	**wolfTLSv1_1_server_method**(void)The wolfTLSv1_1_server_method() function is used to indicate that the application is a server and will only support the TLS 1.1 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().
WOLFSSL_METHOD *	**wolfTLSv1_1_client_method**(void)The wolfTLSv1_1_client_method() function is used to indicate that the application is a client and will only support the TLS 1.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

	Name
WOLFSSL_METHOD *	**wolfTLSv1_2_server_method**(void)The wolfTLSv1_2_server_method() function is used to indicate that the application is a server and will only support the TLS 1.2 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().
WOLFSSL_METHOD *	**wolfTLSv1_2_client_method**(void)The wolfTLSv1_2_client_method() function is used to indicate that the application is a client and will only support the TLS 1.2 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().
WOLFSSL_METHOD *	**wolfDTLSv1_client_method**(void)The wolfDTLSv1_client_method() function is used to indicate that the application is a client and will only support the DTLS 1.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). This function is only available when wolfSSL has been compiled with DTLS support (–enable_dtls, or by defining wolfSSL_DTLS).
WOLFSSL_METHOD *	**wolfDTLSv1_server_method**(void)The wolfDTLSv1_server_method() function is used to indicate that the application is a server and will only support the DTLS 1.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). This function is only available when wolfSSL has been compiled with DTLS support (–enable_dtls, or by defining wolfSSL_DTLS).
WOLFSSL_METHOD *	**wolfDTLSv1_3_server_method**(void)The wolfDTLSv1_3_server_method() function is used to indicate that the application is a server and will only support the DTLS 1.3 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). This function is only available when wolfSSL has been compiled with DTLSv1.3 support (–enable_dtls13, or by defining wolfSSL_DTLS13).

	Name
WOLFSSL_METHOD *	**wolfDTLSv1_3_client_method**(void)The wolfDTLSv1_3_client_method() function is used to indicate that the application is a client and will only support the DTLS 1.3 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). This function is only available when wolfSSL has been compiled with DTLSv1.3 support (–enable_dtls13, or by defining wolfSSL_DTLS13).
WOLFSSL_METHOD *	**wolfDTLS_server_method**(void)The wolfDTLS_server_method() function is used to indicate that the application is a server and will support the highest version of DTLS available and all the version up to the minimum version allowed. The default minimum version allowed is based on the define WOLFSSL_MIN_DTLS_DOWNGRADE and can be changed at runtime using wolfSSL_SetMinVersion(). This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). This function is only available when wolfSSL has been compiled with DTLS support (–enable_dtls, or by defining wolfSSL_DTLS).
WOLFSSL_METHOD *	**wolfDTLS_client_method**(void)The wolfDTLS_client_method() function is used to indicate that the application is a client and will support the highest version of DTLS available and all the version up to the minimum version allowed. The default minimum version allowed is based on the define WOLFSSL_MIN_DTLS_DOWNGRADE and can be changed at runtime using wolfSSL_SetMinVersion(). This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). This function is only available when wolfSSL has been compiled with DTLS support (–enable_dtls, or by defining wolfSSL_DTLS).
WOLFSSL_METHOD *	**wolfDTLSv1_2_server_method**(void)This function creates and initializes a WOLFSSL_METHOD for the server side.

	Name
int	**wolfSSL_use_old_poly**(WOLFSSL * ssl, int value)Since there is some differences between the first release and newer versions of chacha-poly AEAD construction we have added an option to communicate with servers/clients using the older version. By default wolfSSL uses the new version.
int	**wolfSSL_dtls_import**(WOLFSSL * ssl, const unsigned char * buf, unsigned int sz)The wolfSSL_dtls_import() function is used to parse in a serialized session state. This allows for picking up the connection after the handshake has been completed. Post-handshake authentication is not available on an imported connection.
int	**wolfSSL_tls_import**(WOLFSSL * ssl, const unsigned char * buf, unsigned int sz)Used to import a serialized TLS session. This function is for importing the state of the connection. A TLS 1.3 session serialized with export version 7 or later carries its traffic and resumption secrets, so a KeyUpdate and the issuing of a NewSessionTicket work on the imported connection; an older serialization restores the record layer alone. WARNING: buf contains sensitive information about the state and is best to be encrypted before storing if stored. Additional debug info can be displayed with the macro WOLFSSL_SESSION_EXPORT_DEBUG defined.
int	**wolfSSL_CTX_dtls_set_export**(WOLFSSL_CTX * ctx, wc_dtls_export func)The wolfSSL_CTX_dtls_set_export() function is used to set the callback function for exporting a session. It is allowed to pass in NULL as the parameter func to clear the export function previously stored. Used on the server side and is called immediately after handshake is completed, for DTLS 1.2 only; a DTLS 1.3 session is exported with wolfSSL_dtls_export().
int	**wolfSSL_dtls_set_export**(WOLFSSL * ssl, wc_dtls_export func)The wolfSSL_dtls_set_export() function is used to set the callback function for exporting a session. It is allowed to pass in NULL as the parameter func to clear the export function previously stored. Used on the server side and is called immediately after handshake is completed, for DTLS 1.2 only; a DTLS 1.3 session is exported with wolfSSL_dtls_export().

	Name
int	**wolfSSL_dtls_export**(WOLFSSL * ssl, unsigned char * buf, unsigned int * sz)The wolfSSL_dtls_export() function is used to serialize a WOLFSSL session into the provided buffer. Allows for less memory overhead than using a function callback for sending a session and choice over when the session is serialized. If buffer is NULL when passed to function then sz will be set to the size of buffer needed for serializing the WOLFSSL session. The export is refused with a negative return before a DTLS 1.3 handshake is done, while a KeyUpdate is in progress, a fragmented message is half sent, a handshake message other than NewSessionTicket is waiting for the peer's ACK, a post-handshake CertificateRequest is unanswered, or a write dup of the object exists. A NewSessionTicket still waiting for its ACK is not exported and is lost. The session tickets themselves are not serialized: any ticket already issued or received is discarded by the export. For a DTLS 1.3 session the blob carries the resumption secret and the ticket nonce, so an imported server can still issue a NewSessionTicket that resumes; use wolfSSL_get1_session before exporting to keep a received ticket.
int	**wolfSSL_tls_export**(WOLFSSL * ssl, unsigned char * buf, unsigned int * sz)Used to export a serialized TLS session. This function is for exporting a serialized state of the connection. In most cases wolfSSL_get1_session should be used instead of wolfSSL_tls_export. The export of a TLS 1.3 session is refused with a negative return before the handshake is done or while a KeyUpdate response is pending. The session tickets themselves are not serialized: any ticket already issued or received is discarded by the export. A TLS 1.3 blob carries the resumption secret and the ticket nonce, so an imported server can still issue a NewSessionTicket that resumes; use wolfSSL_get1_session before exporting to keep a received ticket. Additional debug info can be displayed with the macro WOLFSSL_SESSION_EXPORT_DEBUG defined. WARNING: buf contains sensitive information about the state and is best to be encrypted before storing if stored.

COPYRIGHT ©2024 wolfSSL Inc.

	Name
int	**wolfSSL_CTX_load_static_memory**(WOLFSSL_CTX ** ctx, wolfSSL_method_func method, unsigned char * buf, unsigned int sz, int flag, int max)This function is used to set aside static memory for a CTX. Memory set aside is then used for the CTX's lifetime and for any SSL objects created from the CTX. By passing in a NULL ctx pointer and a wolfSSL_method_func function the creation of the CTX itself will also use static memory. wolfSSL_method_func has the function signature of WOLFSSL_METHOD* (*wolfSSL_method_func)(void* heap);. Passing in 0 for max makes it behave as if not set and no max concurrent use restrictions is in place. The flag value passed in determines how the memory is used and behavior while operating. Available flags are the following: 0 - default general memory, WOLFMEM_IO_POOL - used for input/output buffer when sending receiving messages and overrides general memory, so all memory in buffer passed in is used for IO, WOLFMEM_IO_FIXED - same as WOLFMEM_IO_POOL but each SSL now keeps two buffers to themselves for their lifetime, WOLFMEM_TRACK_STATS - each SSL keeps track of memory stats while running.
int	**wolfSSL_CTX_is_static_memory**(WOLFSSL_CTX * ctx, WOLFSSL_MEM_STATS * mem_stats)This function does not change any of the connections behavior and is used only for gathering information about the static memory usage.
int	**wolfSSL_is_static_memory**(WOLFSSL * ssl, WOLFSSL_MEM_CONN_STATS * mem_stats)wolfSSL_is_static_memory is used to gather information about a SSL's static memory usage. The return value indicates if static memory is being used and WOLFSSL_MEM_CONN_STATS will be filled out if and only if the flag WOLFMEM_TRACK_STATS was passed to the parent CTX when loading in static memory.
int	**wolfSSL_CTX_use_certificate_file**(WOLFSSL_CTX * ctx, const char * file, int format)This function loads a certificate file into the SSL context (WOLFSSL_CTX). The file is provided by the file argument. The format argument specifies the format type of the file, either SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. Please see the examples for proper usage.

COPYRIGHT ©2024 wolfSSL Inc.

	Name
int	**wolfSSL_CTX_use_PrivateKey_file**(WOLFSSL_CTX * ctx, const char * file, int format)This function loads a private key file into the SSL context (WOLFSSL_CTX). The file is provided by the file argument. The format argument specifies the format type of the file - SSL_FILETYPE_ASN1or SSL_FILETYPE_PEM. Please see the examples for proper usage.
int	**wolfSSL_CTX_load_verify_locations**(WOLFSSL_CTX * ctx, const char * file, const char * path)This function loads PEM_formatted CA certificate files into the SSL context (WOLFSSL_CTX). These certificates will be treated as trusted root certificates and used to verify certs received from peers during the SSL handshake. The root certificate file, provided by the file argument, may be a single certificate or a file containing multiple certificates. If multiple CA certs are included in the same file, wolfSSL will load them in the same order they are presented in the file. The path argument is a pointer to the name of a directory that contains certificates of trusted root CAs. If the value of file is not NULL, path may be specified as NULL if not needed. If path is specified and NO_WOLFSSL_DIR was not defined when building the library, wolfSSL will load all CA certificates located in the given directory. This function will attempt to load all files in the directory. This function expects PEM formatted CERT_TYPE file with header “—–BEGIN CERTIFICATE—–”.

	Name
int	**wolfSSL_CTX_load_verify_locations_ex**(WOLFSSL_CTX * ctx, const char * file, const char * path, word32 flags)This function loads PEM_formatted CA certificate files into the SSL context (WOLFSSL_CTX). These certificates will be treated as trusted root certificates and used to verify certs received from peers during the SSL handshake. The root certificate file, provided by the file argument, may be a single certificate or a file containing multiple certificates. If multiple CA certs are included in the same file, wolfSSL will load them in the same order they are presented in the file. The path argument is a pointer to the name of a directory that contains certificates of trusted root CAs. If the value of file is not NULL, path may be specified as NULL if not needed. If path is specified and NO_WOLFSSL_DIR was not defined when building the library, wolfSSL will load all CA certificates located in the given directory. This function will attempt to load all files in the directory based on flags specified. This function expects PEM formatted CERT_TYPE files with header "—–BEGIN CERTIFICATE—–".
const char **	**wolfSSL_get_system_CA_dirs**(word32 * num)This function returns a pointer to an array of strings representing directories wolfSSL will search for system CA certs when wolfSSL_CTX_load_system_CA_certs is called. On systems that don't store certificates in an accessible system directory (such as Apple platforms), this function will always return NULL.
int	**wolfSSL_CTX_load_system_CA_certs**(WOLFSSL_CTX * ctx)On most platforms (including Linux and Windows), this function attempts to load CA certificates into a WOLFSSL_CTX from an OS-dependent CA certificate store. Loaded certificates will be trusted.
int	**wolfSSL_CTX_trust_peer_cert**(WOLFSSL_CTX * ctx, const char * file, int type)This function loads a certificate to use for verifying a peer when performing a TLS/SSL handshake. The peer certificate sent during the handshake is compared by using the SKID when available and the signature. If these two things do not match then any loaded CAs are used. Feature is enabled by defining the macro WOLFSSL_TRUST_PEER_CERT. Please see the examples for proper usage.

	Name
int	**wolfSSL_CTX_use_certificate_chain_file**(WOLFSSL_CTX * ctx, const char * file)This function loads a chain of certificates into the SSL context (WOLFSSL_CTX). The file containing the certificate chain is provided by the file argument, and must contain PEM_formatted certificates. This function will process up to MAX_CHAIN_DEPTH (default = 9, defined in internal.h) certificates, plus the subject cert.
int	**wolfSSL_CTX_use_RSAPrivateKey_file**(WOLFSSL_CTX * ctx, const char * file, int format)This function loads the private RSA key used in the SSL connection into the SSL context (WOLFSSL_CTX). This function is only available when wolfSSL has been compiled with the OpenSSL compatibility layer enabled (–enable_opensslExtra, #define OPENSSL_EXTRA), and is identical to the more_typically used wolfSSL_CTX_use_PrivateKey_file() function. The file argument contains a pointer to the RSA private key file, in the format specified by format.
long	**wolfSSL_get_verify_depth**(WOLFSSL * ssl)This function returns the maximum chain depth allowed, which is 9 by default, for a valid session i.e. there is a non_null session object (ssl).
long	**wolfSSL_CTX_get_verify_depth**(WOLFSSL_CTX * ctx)This function gets the certificate chaining depth using the CTX structure.
int	**wolfSSL_use_certificate_file**(WOLFSSL * ssl, const char * file, int format)This function loads a certificate file into the SSL session (WOLFSSL structure). The certificate file is provided by the file argument. The format argument specifies the format type of the file - either SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.
int	**wolfSSL_use_PrivateKey_file**(WOLFSSL * ssl, const char * file, int format)This function loads a private key file into the SSL session (WOLFSSL structure). The key file is provided by the file argument. The format argument specifies the format type of the file - SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

	Name
int	**wolfSSL_use_certificate_chain_file**(WOLFSSL * ssl, const char * file)This function loads a chain of certificates into the SSL session (WOLFSSL structure). The file containing the certificate chain is provided by the file argument, and must contain PEM_formatted certificates. This function will process up to MAX_CHAIN_DEPTH (default = 9, defined in internal.h) certificates, plus the subject certificate.
int	**wolfSSL_use_RSAPrivateKey_file**(WOLFSSL * ssl, const char * file, int format)This function loads the private RSA key used in the SSL connection into the SSL session (WOLFSSL structure). This function is only available when wolfSSL has been compiled with the OpenSSL compatibility layer enabled (–enable_opensslExtra, #define OPENSSL_EXTRA), and is identical to the more_typically used wolfSSL_use_PrivateKey_file() function. The file argument contains a pointer to the RSA private key file, in the format specified by format.
int	**wolfSSL_CTX_der_load_verify_locations**(WOLFSSL_CTX * ctx, const char * file, int format)This function is similar to wolfSSL_CTX_load_verify_locations, but allows the loading of DER_formatted CA files into the SSL context (WOLFSSL_CTX). It may still be used to load PEM_formatted CA files as well. These certificates will be treated as trusted root certificates and used to verify certs received from peers during the SSL handshake. The root certificate file, provided by the file argument, may be a single certificate or a file containing multiple certificates. If multiple CA certs are included in the same file, wolfSSL will load them in the same order they are presented in the file. The format argument specifies the format which the certificates are in either, SSL_FILETYPE_PEM or SSL_FILETYPE_ASN1 (DER). Unlike wolfSSL_CTX_load_verify_locations, this function does not allow the loading of CA certificates from a given directory path. Note that this function is only available when the wolfSSL library was compiled with WOLFSSL_DER_LOAD defined.
WOLFSSL_CTX *	**wolfSSL_CTX_new**(WOLFSSL_METHOD *)This function creates a new SSL context, taking a desired SSL/TLS protocol method for input.
WOLFSSL *	**wolfSSL_new**(WOLFSSL_CTX *)This function creates a new SSL session, taking an already created SSL context as input.

	Name
int	**wolfSSL_set_fd**(WOLFSSL * ssl, int fd)This function assigns a file descriptor (fd) as the input/output facility for the SSL connection. Typically this will be a socket file descriptor.
int	**wolfSSL_set_dtls_fd_connected**(WOLFSSL * ssl, int fd)This function assigns a file descriptor (fd) as the input/output facility for the SSL connection. Typically this will be a socket file descriptor. This is a DTLS specific API because it marks that the socket is connected. recvfrom and sendto calls on this fd will have the addr and addr_len parameters set to NULL.

	Name
int	**wolfDTLS_SetChGoodCb**(WOLFSSL * ssl, ClientHelloGoodCb cb, void * user_ctx)Allows setting a callback for a correctly processed and verified DTLS client hello. When using a cookie exchange mechanism (either the HelloVerifyRequest in DTLS 1.2 or the HelloRetryRequest with a cookie extension in DTLS 1.3) this callback is called after the cookie exchange has succeeded. This is useful to use one WOLFSSL object as the listener for new connections and being able to isolate the WOLFSSL object once the ClientHello is verified (either through a cookie exchange or just checking if the ClientHello had the correct format). With DTLS cookies disabled, the callback is invoked once per ClientHello, only after complete successful ClientHello processing, not on its first fragment. Retries, retransmitted fragments and a second ClientHello after a key_share HelloRetryRequest do not repeat the notification. The object is already stateful before input is read, but the peer's return_routability has not been verified. Returning 0 accepts the ClientHello and the handshake continues. Any negative return stops the accept call with that error and leaves the accept state machine at the ClientHello, so nothing is sent to the peer and the next accept call asks the callback again instead of continuing the handshake. All negative returns behave this way; the value only decides what wolfSSL_get_error() reports, so what it means is the application's choice. A callback that is not ready to decide can return WANT_READ or WANT_WRITE and be asked again as many times as it needs, and one that refuses the peer returns its own error and keeps the server silent by refusing again on every later accept call. The cookie mode cannot be changed from the callback; see wolfSSL_disable_cookie(). DTLS 1.2: https://datatracker.ietf.org/doc/html/rfc6347#section_4.2.1 DTLS 1.3: https://www.rfc_editor.org/rfc/rfc8446#section_4.2.2.
char *	**wolfSSL_get_cipher_list**(int priority)Get the name of cipher at priority level passed in.
int	**wolfSSL_get_ciphers**(char * buf, int len)This function gets the ciphers enabled in wolfSSL.
const char *	**wolfSSL_get_cipher_name**(WOLFSSL * ssl)This function gets the cipher name in the format DHE-RSA by passing through argument to wolfSSL_get_cipher_name_internal.

	Name
int	**wolfSSL_get_fd**(const WOLFSSL * ssl)This function returns the read file descriptor (fd) used as the input facility for the SSL connection. Typically this will be a socket file descriptor.
int	**wolfSSL_get_wfd**(const WOLFSSL * ssl)This function returns the write file descriptor (fd) used as the output facility for the SSL connection. Typically this will be a socket file descriptor.
void	**wolfSSL_set_using_nonblock**(WOLFSSL * ssl, int nonblock)This function informs the WOLFSSL object that the underlying I/O is non_blocking. After an application creates a WOLFSSL object, if it will be used with a non_blocking socket, call wolfSSL_set_using_nonblock() on it. This lets the WOLFSSL object know that receiving EWOULDBLOCK means that the recvfrom call would block rather than that it timed out.
int	**wolfSSL_get_using_nonblock**(WOLFSSL *)This function allows the application to determine if wolfSSL is using non_blocking I/O. If wolfSSL is using non_blocking I/O, this function will return 1, otherwise 0. After an application creates a WOLFSSL object, if it will be used with a non_blocking socket, call wolfSSL_set_using_nonblock() on it. This lets the WOLFSSL object know that receiving EWOULDBLOCK means that the recvfrom call would block rather than that it timed out.

	Name
int	**wolfSSL_write**(WOLFSSL * ssl, const void * data, int sz)This function writes sz bytes from the buffer, data, to the SSL connection, ssl. If necessary, wolfSSL_write() will negotiate an SSL/TLS session if the handshake has not already been performed yet by wolfSSL_connect() or wolfSSL_accept(). When using (D)TLSv1.3 and early data feature is compiled in, this function progresses the handshake only up to the point when it is possible to send data. Next invocations of wolfSSL_Connect()/wolfSSL_Accept()/wolfSSL_read() will complete the handshake. wolfSSL_write() works with both blocking and non_blocking I/O. When the underlying I/O is non_blocking, wolfSSL_write() will return when the underlying I/O could not satisfy the needs of wolfSSL_write() to continue. In this case, a call to wolfSSL_get_error() will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_write() when the underlying I/O is ready. If the underlying I/O is blocking, wolfSSL_write() will only return once the buffer data of size sz has been completely written or an error occurred.
int	**wolfSSL_write_ex**(WOLFSSL * ssl, const void * data, size_t sz, size_t * wr)This function writes sz bytes from the buffer, data, to the SSL connection, ssl, and reports the number of bytes written. It is equivalent to wolfSSL_write() except that the length written is returned through wr and the return value only indicates success or failure. Whether a partial write counts as success depends on WOLFSSL_MODE_ENABLE_PARTIAL_WRITE having been set with wolfSSL_CTX_set_mode(); without it, anything short of the full length is a failure.

	Name
int	**wolfSSL_read**(WOLFSSL * ssl, void * data, int sz)This function reads sz bytes from the SSL session (ssl) internal read buffer into the buffer data. The bytes read are removed from the internal receive buffer. If necessary wolfSSL_read() will negotiate an SSL/TLS session if the handshake has not already been performed yet by wolfSSL_connect() or wolfSSL_accept(). The SSL/TLS protocol uses SSL records which have a maximum size of 16kB (the max record size can be controlled by the MAX_RECORD_SIZE define in /wolfssl/internal.h). As such, wolfSSL needs to read an entire SSL record internally before it is able to process and decrypt the record. Because of this, a call to wolfSSL_read() will only be able to return the maximum buffer size which has been decrypted at the time of calling. There may be additional not_yet_decrypted data waiting in the internal wolfSSL receive buffer which will be retrieved and decrypted with the next call to wolfSSL_read(). If sz is larger than the number of bytes in the internal read buffer, SSL_read() will return the bytes available in the internal read buffer. If no bytes are buffered in the internal read buffer yet, a call to wolfSSL_read() will trigger processing of the next record.
int	**wolfSSL_read_ex**(WOLFSSL * ssl, void * data, size_t sz, size_t * rd)This function reads up to sz bytes of decrypted application data from the SSL connection, ssl, into the buffer, data, and reports the number of bytes read. It is equivalent to wolfSSL_read() except that the length read is returned through rd and the return value only indicates whether any application data was read.

	Name
int	**wolfSSL_peek**(WOLFSSL * ssl, void * data, int sz)This function copies sz bytes from the SSL session (ssl) internal read buffer into the buffer data. This function is identical to wolfSSL_read() except that the data in the internal SSL session receive buffer is not removed or modified. If necessary, like wolfSSL_read(), wolfSSL_peek() will negotiate an SSL/TLS session if the handshake has not already been performed yet by wolfSSL_connect() or wolfSSL_accept(). The SSL/TLS protocol uses SSL records which have a maximum size of 16kB (the max record size can be controlled by the MAX_RECORD_SIZE define in /wolfssl/internal.h). As such, wolfSSL needs to read an entire SSL record internally before it is able to process and decrypt the record. Because of this, a call to wolfSSL_peek() will only be able to return the maximum buffer size which has been decrypted at the time of calling. There may be additional not_yet_decrypted data waiting in the internal wolfSSL receive buffer which will be retrieved and decrypted with the next call to wolfSSL_peek() / wolfSSL_read(). If sz is larger than the number of bytes in the internal read buffer, SSL_peek() will return the bytes available in the internal read buffer. If no bytes are buffered in the internal read buffer yet, a call to wolfSSL_peek() will trigger processing of the next record.
int	**wolfSSL_accept**(WOLFSSL * ssl)This function is called on the server side and waits for an SSL client to initiate the SSL/TLS handshake. When this function is called, the underlying communication channel has already been set up. wolfSSL_accept() works with both blocking and non_blocking I/O. When the underlying I/O is non_blocking, wolfSSL_accept() will return when the underlying I/O could not satisfy the needs of wolfSSL_accept to continue the handshake. In this case, a call to wolfSSL_get_error() will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_accept when data is available to read and wolfSSL will pick up where it left off. When using a non_blocking socket, nothing needs to be done, but select() can be used to check for the required condition. If the underlying I/O is blocking, wolfSSL_accept() will only return once the handshake has been finished or an error occurred.

	Name
int	**wolfDTLS_accept_stateless**(WOLFSSL * ssl)This function statelessly listens for an SSL client to initiate the DTLS handshake. Cookies must be enabled. A cookie_disabled DTLS object is rejected before I/O or callback changes: WOLFSSL_FATAL_ERROR is returned and wolfSSL_get_error() reports BAD_STATE_E. Use wolfSSL_accept() for cookie-disabled connections instead.
int	**wolfSSL_disable_cookie**(WOLFSSL * ssl)Disable server cookies for DTLS 1.2, DTLS 1.3 and TLS 1.3, including DTLS 1.2 fallback from DTLS 1.3. Cookies are enabled by default for DTLS. For DTLS, ordinary wolfSSL_accept() or wolfSSL_accept_TLSv13() commits it to stateful processing before the first read, even if a nonblocking accept has no input available. The application must isolate/demultiplex the peer before accepting; disabling cookies removes return_routability verification and exposes the server to DoS/amplification attacks. wolfDTLS_accept_stateless() cannot be used.
int	**wolfSSL_enable_cookie**(WOLFSSL * ssl)Enable server cookies for DTLS 1.2, DTLS 1.3 and TLS 1.3. Like wolfSSL_disable_cookie(), this does not change stateful processing. It can undo a disable before accept begins, including wolfSSL_disable_hrr_cookie(). Missing primary cookie secrets are randomly generated immediately; existing primary and secondary secrets are preserved.
void	**wolfSSL_CTX_free**(WOLFSSL_CTX * ctx)This function frees an allocated WOLFSSL_CTX object. This function decrements the CTX reference count and only frees the context when the reference count has reached 0.
void	**wolfSSL_free**(WOLFSSL * ssl)This function frees an allocated wolfSSL object.

COPYRIGHT ©2024 wolfSSL Inc.

	Name
int	**wolfSSL_shutdown**(WOLFSSL * ssl)This function shuts down an active SSL/TLS connection using the SSL session, ssl. This function will try to send a “close notify” alert to the peer. The calling application can choose to wait for the peer to send its “close notify” alert in response or just go ahead and shut down the underlying connection after directly calling wolfSSL_shutdown (to save resources). Either option is allowed by the TLS specification. If the underlying connection will be used again in the future, the complete two_directional shutdown procedure must be performed to keep synchronization intact between the peers. wolfSSL_shutdown() works with both blocking and non_blocking I/O. When the underlying I/O is non_blocking, wolfSSL_shutdown() will return an error if the underlying I/O could not satisfy the needs of wolfSSL_shutdown() to continue. In this case, a call to wolfSSL_get_error() will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_shutdown() when the underlying I/O is ready.
int	**wolfSSL_SendUserCanceled**(WOLFSSL * ssl)This function sends a user_canceled alert to the peer and then shuts the connection down by calling wolfSSL_shutdown(). It is used when the application abandons a connection for its own reasons rather than because of a protocol failure.

	Name
int	**wolfSSL_send**(WOLFSSL * ssl, const void * data, int sz, int flags)This function writes sz bytes from the buffer, data, to the SSL connection, ssl, using the specified flags for the underlying write operation. If necessary wolfSSL_send() will negotiate an SSL/TLS session if the handshake has not already been performed yet by wolfSSL_connect() or wolfSSL_accept(). wolfSSL_send() works with both blocking and non_blocking I/O. When the underlying I/O is non_blocking, wolfSSL_send() will return when the underlying I/O could not satisfy the needs of wolfSSL_send to continue. In this case, a call to wolfSSL_get_error() will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_send() when the underlying I/O is ready. If the underlying I/O is blocking, wolfSSL_send() will only return once the buffer data of size sz has been completely written or an error occurred.

COPYRIGHT ©2024 wolfSSL Inc.

	Name
int	**wolfSSL_recv**(WOLFSSL * ssl, void * data, int sz, int flags)This function reads sz bytes from the SSL session (ssl) internal read buffer into the buffer data using the specified flags for the underlying recv operation. The bytes read are removed from the internal receive buffer. This function is identical to wolfSSL_read() except that it allows the application to set the recv flags for the underlying read operation. If necessary wolfSSL_recv() will negotiate an SSL/TLS session if the handshake has not already been performed yet by wolfSSL_connect() or wolfSSL_accept(). The SSL/TLS protocol uses SSL records which have a maximum size of 16kB (the max record size can be controlled by the MAX_RECORD_SIZE define in /wolfssl/internal.h). As such, wolfSSL needs to read an entire SSL record internally before it is able to process and decrypt the record. Because of this, a call to wolfSSL_recv() will only be able to return the maximum buffer size which has been decrypted at the time of calling. There may be additional not_yet_decrypted data waiting in the internal wolfSSL receive buffer which will be retrieved and decrypted with the next call to wolfSSL_recv(). If sz is larger than the number of bytes in the internal read buffer, SSL_recv() will return the bytes available in the internal read buffer. If no bytes are buffered in the internal read buffer yet, a call to wolfSSL_recv() will trigger processing of the next record.
int	**wolfSSL_get_error**(WOLFSSL * ssl, int ret)This function returns a unique error code describing why the previous API function call (wolfSSL_connect, wolfSSL_accept, wolfSSL_read, wolfSSL_write, etc.) resulted in an error return code (SSL_FAILURE). The return value of the previous function is passed to wolfSSL_get_error through ret. After wolfSSL_get_error is called and returns the unique error code, wolfSSL_ERR_error_string() may be called to get a human_readable error string. See wolfSSL_ERR_error_string() for more information.
int	**wolfSSL_get_alert_history**(WOLFSSL * ssl, WOLFSSL_ALERT_HISTORY * h)This function gets the alert history.

	Name
int	**wolfSSL_set_session**(WOLFSSL * ssl, WOLFSSL_SESSION * session)This function sets the session to be used when the SSL object, ssl, is used to establish a SSL/TLS connection. For session resumption, before calling wolfSSL_shutdown() with your session object, an application should save the session ID from the object with a call to wolfSSL_get1_session(), which returns a pointer to the session. Later, the application should create a new WOLFSSL object and assign the saved session with wolfSSL_set_session(). At this point, the application may call wolfSSL_connect() and wolfSSL will try to resume the session. The wolfSSL server code allows session resumption by default. The object returned by wolfSSL_get1_session() needs to be freed after the application is done with it by calling wolfSSL_SESSION_free() on it.
WOLFSSL_SESSION *	**wolfSSL_get_session**(WOLFSSL * ssl)Returns a pointer to the current session (WOLFSSL_SESSION) used in ssl. This is a non_persistent pointer to the WOLFSSL_SESSION object, freed when wolfSSL_free is called, so the call should only be used to inspect or modify the current session. For session resumption use wolfSSL_get1_session(), which returns a reference the caller owns and releases with wolfSSL_SESSION_free().
void	**wolfSSL_flush_sessions**(WOLFSSL_CTX * ctx, long tm)This function flushes session from the session cache which have expired. The time, tm, is used for the time comparison. Note that wolfSSL currently uses a static table for sessions, so no flushing is needed. As such, this function is currently just a stub. This function provides OpenSSL compatibility (SSL_flush_sessions) when wolfSSL is compiled with the OpenSSL compatibility layer.
int	**wolfSSL_SetServerID**(WOLFSSL * ssl, const unsigned char * id, int len, int newSession)This function associates the client session with the server id. If the newSession flag is on, an existing session won't be reused.
int	**wolfSSL_GetSessionIndex**(WOLFSSL * ssl)This function gets the session index of the WOLFSSL structure.

	Name
int	**wolfSSL_GetSessionAtIndex**(int index, WOLFSSL_SESSION * session)This function gets the session at specified index of the session cache and copies it into memory. The WOLFSSL_SESSION structure holds the session information. The copy is independent of the cache entry: it does not share the ticket buffer, peer certificate or ex_data with the cache, and it stays valid after the cache entry is overwritten or evicted. The caller owns the copy and releases it with wolfSSL_SESSION_free().
WOLFSSL_X509_CHAIN *	**wolfSSL_SESSION_get_peer_chain**(WOLFSSL_SESSION * session)Returns the peer certificate chain from the WOLFSSL_SESSION struct.

	Name
void	**wolfSSL_CTX_set_verify**(WOLFSSL_CTX * ctx, int mode, VerifyCallback verify_callback)This function sets the verification method for remote peers and also allows a verify callback to be registered with the SSL context. The verify callback will be called only when a verification failure has occurred. If no verify callback is desired, the NULL pointer can be used for verify_callback. The verification mode of peer certificates is a logically OR'd list of flags. The possible flag values include: SSL_VERIFY_NONE Client mode: the client will not verify the certificate received from the server and the handshake will continue as normal. Server mode: the server will not send a certificate request to the client. As such, client verification will not be enabled. SSL_VERIFY_PEER Client mode: the client will verify the certificate received from the server during the handshake. This is turned on by default in wolfSSL, therefore, using this option has no effect. Server mode: the server will send a certificate request to the client and verify the client certificate received. SSL_VERIFY_FAIL_IF_NO_PEER_CERT Client mode: no effect when used on the client side. Server mode: the verification will fail on the server side if the client fails to send a certificate when requested to do so (when using SSL_VERIFY_PEER on the SSL server). SSL_VERIFY_FAIL_EXCEPT_PSK Client mode: no effect when used on the client side. Server mode: the verification is the same as SSL_VERIFY_FAIL_IF_NO_PEER_CERT except in the case of a PSK connection. If a PSK connection is being made then the connection will go through without a peer cert. SSL_VERIFY_CLIENT_ONCE Accepted and ignored. It is present so that OpenSSL_derived code compiles unchanged; wolfSSL does not act on it, does not store it, and wolfSSL_get_verify_mode() / wolfSSL_CTX_get_verify_mode() will not report it back. Including it in mode has no effect on any of the other flags.

COPYRIGHT ©2024 wolfSSL Inc.

	Name
void	**wolfSSL_set_verify**(WOLFSSL * ssl, int mode, VerifyCallback verify_callback)This function sets the verification method for remote peers and also allows a verify callback to be registered with the SSL session. The verify callback will be called only when a verification failure has occurred. If no verify callback is desired, the NULL pointer can be used for verify_callback. The verification mode of peer certificates is a logically OR'd list of flags. The possible flag values include: SSL_VERIFY_NONE Client mode: the client will not verify the certificate received from the server and the handshake will continue as normal. Server mode: the server will not send a certificate request to the client. As such, client verification will not be enabled. SSL_VERIFY_PEER Client mode: the client will verify the certificate received from the server during the handshake. This is turned on by default in wolfSSL, therefore, using this option has no effect. Server mode: the server will send a certificate request to the client and verify the client certificate received. SSL_VERIFY_FAIL_IF_NO_PEER_CERT Client mode: no effect when used on the client side. Server mode: the verification will fail on the server side if the client fails to send a certificate when requested to do so (when using SSL_VERIFY_PEER on the SSL server). SSL_VERIFY_FAIL_EXCEPT_PSK Client mode: no effect when used on the client side. Server mode: the verification is the same as SSL_VERIFY_FAIL_IF_NO_PEER_CERT except in the case of a PSK connection. If a PSK connection is being made then the connection will go through without a peer cert. SSL_VERIFY_CLIENT_ONCE Accepted and ignored. It is present so that OpenSSL_derived code compiles unchanged; wolfSSL does not act on it, does not store it, and wolfSSL_get_verify_mode() / wolfSSL_CTX_get_verify_mode() will not report it back. Including it in mode has no effect on any of the other flags.
void	**wolfSSL_SetCertCbCtx**(WOLFSSL * ssl, void * ctx)This function stores user CTX object information for verify callback.
void	**wolfSSL_CTX_SetCertCbCtx**(WOLFSSL_CTX * ctx, void * userCtx)This function stores user CTX object information for verify callback.

	Name
int	**wolfSSL_CTX_SetChainVerifyCb**(WOLFSSL_CTX * ctx, ChainVerifyCb cb)Replaces wolfSSL's verification of the peer's certificate chain with an application callback, for every SSL/TLS object created from the context. When a callback is set, wolfSSL decodes the certificates from the Certificate message and hands them to the callback as raw DER, the peer's own certificate first. It builds no chain, verifies no signature and checks no date, revocation status, key usage or host name, and the verify callback set with wolfSSL_CTX_set_verify() is not called; the callback is consulted for any certificate the peer sends, even under WOLFSSL_VERIFY_NONE. A server only requests a client certificate under WOLFSSL_VERIFY_PEER and only requires one with WOLFSSL_VERIFY_FAIL_IF_NO_PEER_CERT, so a server that relies on the callback for client authentication must set both. Malformed DER, a certificate the parser refuses regardless of the verify mode such as one with a zero serial number, or one whose decoding stops part way such as one with an unsupported key algorithm, still fails the handshake before the callback is called. An unknown critical extension or a key usage inconsistent with the basic constraints is left for the callback to judge. The callback returns 0 to accept, CHAIN_VERIFY_WANT_E to suspend the handshake until the application re_enters wolfSSL_connect(), wolfSSL_accept(), wolfSSL_read() or wolfSSL_write() (a certificate received after the handshake is resumed by wolfSSL_read() only, and wolfSSL_write() fails with CHAIN_VERIFY_WANT_E until then), or any other value to reject with CHAIN_VERIFY_CB_E and a fatal bad_certificate alert; 0 is the only accepting value, so returning WOLFSSL_SUCCESS rejects. Because wolfSSL checks nothing itself, wolfSSL_get_verify_result() reports WOLFSSL_X509_V_OK once the callback accepts, recording its verdict rather than a verification wolfSSL performed. DTLS, raw public keys and verifying a stapled OCSP response are not supported with the callback: setting it on a context configured for one of them fails, and so does the handshake of a connection using one of them, with CHAIN_VERIFY_UNSUPPORTED_E. The stapling check covers the client, the side that verifies a stapled response; a server may still staple its own status. Requires WOLFSSL_CHAIN_VERIFY_CB (–enable_chain_verify_cb).

	Name
int	**wolfSSL_SetChainVerifyCb**(WOLFSSL * ssl, ChainVerifyCb cb)Sets the chain verification callback for one SSL/TLS object. It takes precedence over the callback set on the context with wolfSSL_CTX_SetChainVerifyCb(), which documents the callback's contract.
void	**wolfSSL_CTX_SetChainVerifyCtx**(WOLFSSL_CTX * ctx, void * userCtx)Sets the user context passed to the chain verification callback for every SSL/TLS object created from the context.
void	**wolfSSL_SetChainVerifyCtx**(WOLFSSL * ssl, void * ctx)Sets the user context passed to the chain verification callback for one SSL/TLS object. It takes precedence over the value set on the context with wolfSSL_CTX_SetChainVerifyCtx().
void *	**wolfSSL_GetChainVerifyCtx**(const WOLFSSL * ssl)Returns the user context the chain verification callback is called with for this SSL/TLS object: the value set with wolfSSL_SetChainVerifyCtx() when there is one, otherwise the context's.
int	**wolfSSL_pending**(WOLFSSL * ssl)This function returns the number of bytes which are buffered and available in the SSL object to be read by wolfSSL_read().
void	**wolfSSL_load_error_strings**(void)This function is for OpenSSL compatibility (SSL_load_error_string) only and takes no action.
int	**wolfSSL_library_init**(void)This function is called internally in wolfSSL_CTX_new(). This function is a wrapper around wolfSSL_Init() and exists for OpenSSL compatibility (SSL_library_init) when wolfSSL has been compiled with OpenSSL compatibility layer. wolfSSL_Init() is the more typically-used wolfSSL initialization function.
int	**wolfSSL_SetDevId**(WOLFSSL * ssl, int devId)This function sets the Device Id at the WOLFSSL session level.
int	**wolfSSL_CTX_SetDevId**(WOLFSSL_CTX * ctx, int devId)This function sets the Device Id at the WOLFSSL_CTX context level.
int	**wolfSSL_CTX_GetDevId**(WOLFSSL_CTX * ctx, WOLFSSL * ssl)This function retrieves the Device Id.

	Name
long	**wolfSSL_CTX_set_session_cache_mode**(WOLFSSL_CTX * ctx, long mode)This function enables or disables SSL session caching. Behavior depends on the value used for mode. The following values for mode are available: SSL_SESS_CACHE_OFF- disable session caching. Session caching is turned on by default. SSL_SESS_CACHE_NO_AUTO_CLEAR - Disable auto-flushing of the session cache. Auto-flushing is turned on by default.
void	**wolfSSL_CTX_sess_set_get_cb**(WOLFSSL_CTX * ctx, WOLFSSL_SESSION *()*(WOLFSSL *ssl, const unsigned char* , int, int *) f)This function registers the callback wolfSSL uses to look a session up in an application managed (external) session cache. It is called on the server side when a peer offers a session id to resume, before the internal cache is consulted. Compatibility layer equivalent of SSL_CTX_sess_set_get_cb().
int	**wolfSSL_set_session_secret_cb**(WOLFSSL * ssl, SessionSecretCb cb, void * ctx)This function sets the session secret callback function. The SessionSecretCb type has the signature: int (*SessionSecretCb)(WOLFSSL* ssl, void* secret, int* secretSz, void* ctx). The sessionSecretCb member of the WOLFSSL struct is set to the parameter cb.
int	**wolfSSL_save_session_cache**(const char * fname)This function persists the session cache to file. It doesn't use memsave because of additional memory use.
int	**wolfSSL_restore_session_cache**(const char * fname)This function restores the persistent session cache from file. It does not use memstore because of additional memory use.
int	**wolfSSL_memsave_session_cache**(void * mem, int sz)This function persists session cache to memory.
int	**wolfSSL_memrestore_session_cache**(const void * mem, int sz)This function restores the persistent session cache from memory.
int	**wolfSSL_get_session_cache_memsize**(void)This function returns how large the session cache save buffer should be.
int	**wolfSSL_CTX_save_cert_cache**(WOLFSSL_CTX * ctx, const char * fname)This function writes the cert cache from memory to file.
int	**wolfSSL_CTX_restore_cert_cache**(WOLFSSL_CTX * ctx, const char * fname)This function persistes certificate cache from a file.

	Name
int	**wolfSSL_CTX_memsave_cert_cache**(WOLFSSL_CTX * ctx, void * mem, int sz, int * used)This function persists the certificate cache to memory.
int	**wolfSSL_CTX_memrestore_cert_cache**(WOLFSSL_CTX * ctx, const void * mem, int sz)This function restores the certificate cache from memory.
int	**wolfSSL_CTX_get_cert_cache_memsize**(WOLFSSL_CTX * ctx)Returns the size the certificate cache save buffer needs to be.
int	**wolfSSL_CTX_set_cipher_list**(WOLFSSL_CTX * ctx, const char * list)This function sets the cipher suite list for a given WOLFSSL_CTX. The list becomes the default cipher suite list for any new WOLFSSL sessions created from the context, and the order in the string is the local preference order from highest to lowest. Each call replaces the previous list. The list is a null-terminated, colon-delimited text string of suite names and/or OpenSSL-style group keywords, for example “TLS13-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:DHE-RSA-AES256-SHA256”.
int	**wolfSSL_set_cipher_list**(WOLFSSL * ssl, const char * list)This function sets the cipher suite list for a given WOLFSSL session. The list format and the set of recognized suite names and keywords are identical to those documented for wolfSSL_CTX_set_cipher_list(); refer to that function for the full keyword/suite tables and required build options. Each call replaces the previous list on the session.
void	**wolfSSL_dtls_set_using_nonblock**(WOLFSSL * ssl, int nonblock)This function informs the WOLFSSL DTLS object that the underlying UDP I/O is non_blocking. After an application creates a WOLFSSL object, if it will be used with a non_blocking UDP socket, call wolfSSL_dtls_set_using_nonblock() on it. This lets the WOLFSSL object know that receiving EWOULDBLOCK means that the recvfrom call would block rather than that it timed out.

	Name
int	**wolfSSL_dtls_get_using_nonblock**(WOLFSSL * ssl)This function allows the application to determine if wolfSSL is using non_blocking I/O with UDP. If wolfSSL is using non_blocking I/O, this function will return 1, otherwise 0. After an application creates a WOLFSSL object, if it will be used with a non_blocking UDP socket, call wolfSSL_dtls_set_using_nonblock() on it. This lets the WOLFSSL object know that receiving EWOULDBLOCK means that the recvfrom call would block rather than that it timed out. This function is only meaningful to DTLS sessions.
int	**wolfSSL_dtls_get_current_timeout**(WOLFSSL * ssl)This function returns the current timeout value in seconds for the WOLFSSL object. When using non-blocking sockets, something in the user code needs to decide when to check for available recv data and how long it has been waiting. The value returned by this function indicates how long the application should wait.
int	**wolfSSL_dtls13_use_quick_timeout**(WOLFSSL * ssl)This function returns true if the application should setup a quicker timeout. When using non_blocking sockets, something in the user code needs to decide when to check for available data and how long it needs to wait. If this function returns true, it means that the library already detected some disruption in the communication, but it wants to wait for a little longer in case some messages from the other peers are still in flight. Is up to the application to fine tune the value of this timer, a good one may be dtls_get_current_timeout() / 4.
void	**wolfSSL_dtls13_set_send_more_acks**(WOLFSSL * ssl, int value)This function sets whether the library should send ACKs to the other peer immediately when detecting disruption or not. Sending ACKs immediately assures minimum latency but it may consume more bandwidth than necessary. If the application manages the timer by itself and this option is set to 0 then application code can use wolfSSL_dtls13_use_quick_timeout() to determine if it should setup a quicker timeout to send those delayed ACKs.
int	**wolfSSL_dtls_set_timeout_init**(WOLFSSL * ssl, int timeout)This function sets the dtls timeout.
int	**wolfSSL_dtls_set_timeout_max**(WOLFSSL * ssl, int timeout)This function sets the maximum dtls timeout.

COPYRIGHT ©2024 wolfSSL Inc.

	Name
int	**wolfSSL_dtls_got_timeout**(WOLFSSL * ssl)When using non-blocking sockets with DTLS, this function should be called on the WOLFSSL object when the controlling code thinks the transmission has timed out. It performs the actions needed to retry the last transmit, including adjusting the timeout value. If it has been too long, this will return a failure.
int	**wolfSSL_dtls_retransmit**(WOLFSSL * ssl)When using non-blocking sockets with DTLS, this function retransmits the last handshake flight ignoring the expected timeout value and retransmit count. It is useful for applications that are using DTLS and need to manage even the timeout and retry count.
int	**wolfSSL_dtls13_do_scheduled_work**(WOLFSSL * ssl)Sends the DTLS 1.3 work that was scheduled while reading. With WOLFSSL_RW_THREADED the read path never transmits, because that would race the write thread over the output buffer and the sending key schedule, so ACKs, retransmissions and a KeyUpdate the peer asked for are only sent from the write side. An application that reads without writing must call this, from the same thread it uses for writing, or those messages are never sent and the peer keeps retransmitting what it is waiting to have acknowledged. Not for write_dup applications, which drain through wolfSSL_write().
int	**wolfSSL_dtls13_pending_work**(WOLFSSL * ssl)Reports whether the object has DTLS 1.3 work waiting to be sent by wolfSSL_dtls13_do_scheduled_work(). Only meaningful with WOLFSSL_RW_THREADED. The answer is advisory and can change as soon as it is returned. Only work the pump can actually carry out is reported, so a drain loop over the pair terminates; waiting for the peer to acknowledge a key update we sent is not reported, as there is nothing to send for it. A record the pump could only write in part is reported, so that the retry it owes is not lost.
int	**wolfSSL_dtls**(WOLFSSL * ssl)This function is used to determine if the SSL session has been configured to use DTLS.
int	**wolfSSL_dtls_set_peer**(WOLFSSL * ssl, void * peer, unsigned int peerSz)This function sets the DTLS peer, peer (sockaddr_in) with size of peerSz.

	Name
int	**wolfSSL_dtls_set_pending_peer**(WOLFSSL * ssl, void * peer, unsigned int peerSz)This function sets the pending DTLS peer, peer (sockaddr_in) with size of peerSz. This sets the pending peer that will be upgraded to a regular peer when we successfully de-protect the next record. This is useful in scenarios where the peer's address can change to avoid off-path attackers from changing the peer address. This should be used with Connection ID's to allow seamless and safe transition to a new peer address.
int	**wolfSSL_dtls_get_peer**(WOLFSSL * ssl, void * peer, unsigned int * peerSz)This function gets the sockaddr_in (of size peerSz) of the current DTLS peer. The function will compare peerSz to the actual DTLS peer size stored in the SSL session. If the peer will fit into peer, the peer's sockaddr_in will be copied into peer, with peerSz set to the size of peer.
int	**wolfSSL_dtls_get0_peer**(WOLFSSL * ssl, const void ** peer, unsigned int * peerSz)This function gets the sockaddr_in (of size peerSz) of the current DTLS peer. This is a zero_copy alternative to wolfSSL_dtls_get_peer().
char *	**wolfSSL_ERR_error_string**(unsigned long errNumber, char * data)This function converts an error code returned by wolfSSL_get_error() into a more human_readable error string. errNumber is the error code returned by wolfSSL_get_error() and data is the storage buffer which the error string will be placed in. The maximum length of data is 80 characters by default, as defined by MAX_ERROR_SZ is wolfssl/wolfcrypt/error.h.
void	**wolfSSL_ERR_error_string_n**(unsigned long e, char * buf, unsigned long len)This function is a version of wolfSSL_ERR_error_string() where len specifies the maximum number of characters that may be written to buf. Like wolfSSL_ERR_error_string(), this function converts an error code returned from wolfSSL_get_error() into a more human-readable error string. The human-readable string is placed in buf.
int	**wolfSSL_get_shutdown**(const WOLFSSL * ssl)This function checks the shutdown conditions in closeNotify or connReset or sentNotify members of the Options structure. The Options structure is within the WOLFSSL structure.

	Name
int	**wolfSSL_session_reused**(WOLFSSL * ssl)This function returns the resuming member of the options struct. The flag indicates whether or not to reuse a session. If not, a new session must be established.
int	**wolfSSL_is_init_finished**(const WOLFSSL * ssl)This function checks to see if the connection is established.
const char *	**wolfSSL_get_version**(WOLFSSL * ssl)Returns the SSL version being used as a string.
int	**wolfSSL_get_current_cipher_suite**(WOLFSSL * ssl)Returns the current cipher suit an ssl session is using.
WOLFSSL_CIPHER *	**wolfSSL_get_current_cipher**(WOLFSSL * ssl)This function returns a pointer to the current cipher in the ssl session. The cipher is filled in from the session on each call, so call it again after the session changes, e.g. after the handshake.
const char *	**wolfSSL_CIPHER_get_name**(const WOLFSSL_CIPHER * cipher)This function matches the cipher suite in the SSL object with the available suites and returns the string representation.
const char *	**wolfSSL_get_cipher**(WOLFSSL *)This function matches the cipher suite in the SSL object with the available suites.
WOLFSSL_SESSION *	**wolfSSL_get1_session**(WOLFSSL * ssl)This function returns the WOLFSSL_SESSION from the WOLFSSL structure as a reference type. This requires calling wolfSSL_SESSION_free to release the session reference. The WOLFSSL_SESSION pointed to contains all the necessary information required to perform a session resumption and reestablish the connection without a new handshake. For session resumption, before calling wolfSSL_shutdown() with your session object, an application should save the session ID from the object with a call to wolfSSL_get1_session(), which returns a pointer to the session. Later, the application should create a new WOLFSSL object and assign the saved session with wolfSSL_set_session(). At this point, the application may call wolfSSL_connect() and wolfSSL will try to resume the session. The wolfSSL server code allows session resumption by default. The object returned by wolfSSL_get1_session() needs to be freed after the application is done with it by calling wolfSSL_SESSION_free() on it.

	Name
WOLFSSL_METHOD *	**wolfSSLv23_client_method**(void)The wolfSSLv23_client_method() function is used to indicate that the application is a client and will support the highest protocol version supported by the server between SSL 3.0 _ TLS 1.3. This function allocates memory for and initializes a new WOLFSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). Both wolfSSL clients and servers have robust version downgrade capability. If a specific protocol version method is used on either side, then only that version will be negotiated or an error will be returned. For example, a client that uses TLSv1 and tries to connect to a SSLv3 only server will fail, likewise connecting to a TLSv1.1 will fail as well. To resolve this issue, a client that uses the wolfSSLv23_client_method() function will use the highest protocol version supported by the server and downgrade to SSLv3 if needed. In this case, the client will be able to connect to a server running SSLv3 - TLSv1.3.
int	**wolfSSL_BIO_get_mem_data**(WOLFSSL_BIO * bio, void * p)This is used to set a byte pointer to the start of the internal memory buffer.
void	**wolfSSL_BIO_set_init**(WOLFSSL_BIO * bio, int init)This is used to set the init flag of a BIO, indicating whether the BIO has been initialised and is ready for use. Typically called from a custom BIO create callback.
int	**wolfSSL_BIO_get_init**(WOLFSSL_BIO * bio)This is used to retrieve the init flag of a BIO, indicating whether the BIO has been initialised and is ready for use.
long	**wolfSSL_BIO_set_fd**(WOLFSSL_BIO * b, int fd, int flag)Sets the file descriptor for bio to use.
int	**wolfSSL_BIO_set_close**(WOLFSSL_BIO * b, long flag)Sets the close flag, used to indicate that the i/o stream should be closed when the BIO is freed.
WOLFSSL_BIO_METHOD *	**wolfSSL_BIO_s_socket**(void)This is used to get a BIO_SOCKET type WOLFSSL_BIO_METHOD.
int	**wolfSSL_BIO_set_write_buf_size**(WOLFSSL_BIO * b, long size)This is used to set the size of write buffer for a WOLFSSL_BIO. If write buffer has been previously set this function will free it when resetting the size. It is similar to wolfSSL_BIO_reset in that it resets read and write indexes to 0.

	Name
int	**wolfSSL_BIO_make_bio_pair**(WOLFSSL_BIO * b1, WOLFSSL_BIO * b2)This is used to pair two bios together. A pair of bios acts similar to a two way pipe writing to one can be read by the other and vice versa. It is expected that both bios be in the same thread, this function is not thread safe. Freeing one of the two bios removes both from being paired. If a write buffer size was not previously set for either of the bios it is set to a default size of 17000 (WOLFSSL_BIO_SIZE) before being paired.
int	**wolfSSL_BIO_ctrl_reset_read_request**(WOLFSSL_BIO * b)This is used to set the read request flag back to 0.
int	**wolfSSL_BIO_nread0**(WOLFSSL_BIO * bio, char ** buf)This is used to get a buffer pointer for reading from. Unlike wolfSSL_BIO_nread the internal read index is not advanced by the number returned from the function call. Reading past the value returned can result in reading out of array bounds.
int	**wolfSSL_BIO_nread**(WOLFSSL_BIO * bio, char ** buf, int num)This is used to get a buffer pointer for reading from. The internal read index is advanced by the number returned from the function call with buf being pointed to the beginning of the buffer to read from. In the case that less bytes are in the read buffer than the value requested with num the lesser value is returned. Reading past the value returned can result in reading out of array bounds.
int	**wolfSSL_BIO_nwrite**(WOLFSSL_BIO * bio, char ** buf, int num)Gets a pointer to the buffer for writing as many bytes as returned by the function. Writing more bytes to the pointer returned then the value returned can result in writing out of bounds.
int	**wolfSSL_BIO_reset**(WOLFSSL_BIO * bio)Resets bio to an initial state. As an example for type BIO_BIO this resets the read and write index.
int	**wolfSSL_BIO_seek**(WOLFSSL_BIO * bio, int ofs)This function adjusts the file pointer to the offset given. This is the offset from the head of the file.
int	**wolfSSL_BIO_write_filename**(WOLFSSL_BIO * bio, char * name)This is used to set and write to a file. WIll overwrite any data currently in the file and is set to close the file when the bio is freed.

	Name
long	**wolfSSL_BIO_set_mem_eof_return**(WOLFSSL_BIO * bio, int v)This is used to set the end of file value. Common value is -1 so as not to get confused with expected positive values.
long	**wolfSSL_BIO_get_mem_ptr**(WOLFSSL_BIO * bio, WOLFSSL_BUF_MEM ** m)This is a getter function for WOLFSSL_BIO memory pointer.
char *	**wolfSSL_X509_NAME_oneline**(WOLFSSL_X509_NAME * name, char * in, int sz)This function copies the name of the x509 into a buffer.
WOLFSSL_X509_NAME *	**wolfSSL_X509_get_issuer_name**(WOLFSSL_X509 * cert)This function returns the name of the certificate issuer.
WOLFSSL_X509_NAME *	**wolfSSL_X509_get_subject_name**(WOLFSSL_X509 * cert)This function returns the subject member of the WOLFSSL_X509 structure.
int	**wolfSSL_X509_get_isCA**(WOLFSSL_X509 * x509)Checks the isCa member of the WOLFSSL_X509 structure and returns the value.
int	**wolfSSL_X509_NAME_get_text_by_NID**(const WOLFSSL_X509_NAME * name, int nid, char * buf, int len)This function gets the text related to the passed in NID value.
int	**wolfSSL_X509_get_signature_type**(WOLFSSL_X509 * x509)This function returns the value stored in the sigOID member of the WOLFSSL_X509 structure.
void	**wolfSSL_X509_free**(WOLFSSL_X509 * x509)This function frees a WOLFSSL_X509 structure.
int	**wolfSSL_X509_get_signature**(WOLFSSL_X509 * x509, unsigned char * buf, int * bufSz)Gets the X509 signature and stores it in the buffer.
int	**wolfSSL_X509_STORE_add_cert**(WOLFSSL_X509_STORE * store, WOLFSSL_X509 * x509)This function adds a certificate to the WOLFSSL_X509_STRE structure.
WOLFSSL_STACK *	**wolfSSL_X509_STORE_CTX_get_chain**(WOLFSSL_X509_STORE_CT * ctx)This function is a getter function for chain variable in WOLFSSL_X509_STORE_CTX structure. Currently chain is not populated.
int	**wolfSSL_X509_STORE_set_flags**(WOLFSSL_X509_STORE * store, unsigned long flag)This function takes in a flag to change the behavior of the WOLFSSL_X509_STORE structure passed in. An example of a flag used is WOLFSSL_CRL_CHECK. X509_V_FLAG_CRL_CHECK and X509_V_FLAG_CRL_CHECK_ALL enable CRL checking like WOLFSSL_CRL_CHECK and WOLFSSL_CRL_CHECKALL.
const byte *	**wolfSSL_X509_notBefore**(WOLFSSL_X509 * x509)This function the certificate “not before” validity encoded as a byte array.

	Name
const byte *	**wolfSSL_X509_notAfter**(WOLFSSL_X509 * x509)This function the certificate “not after” validity encoded as a byte array.
WOLFSSL_BIGNUM *	**wolfSSL_ASN1_INTEGER_to_BN**(const WOLFSSL_ASN1_INTEGER * ai, WOLFSSL_BIGNUM * bn)This function is used to copy a WOLFSSL_ASN1_INTEGER value to a WOLFSSL_BIGNUM structure.
long	**wolfSSL_CTX_add_extra_chain_cert**(WOLFSSL_CTX * ctx, WOLFSSL_X509 * x509)This function adds the certificate to the internal chain being built in the WOLFSSL_CTX structure.
int	**wolfSSL_CTX_get_read_ahead**(WOLFSSL_CTX * ctx)This function returns the get read ahead flag from a WOLFSSL_CTX structure.
int	**wolfSSL_CTX_set_read_ahead**(WOLFSSL_CTX * ctx, int v)This function sets the read ahead flag in the WOLFSSL_CTX structure. When enabled, the record_header read pulls in up to a full TLS record in a single recv(), so the body (and any following buffered records) is obtained without a second syscall. The flag only changes I/O behaviour when the library is built with read_ahead support (–enable_readahead / WOLFSSL_TLS_READ_AHEAD); otherwise it is stored but inert.
int	**wolfSSL_CTX_set_default_read_buffer_len**(WOLFSSL_CTX * ctx, size_t len)This function sets the read_ahead receive window size for contexts created from this WOLFSSL_CTX. When read_ahead is enabled (wolfSSL_CTX_set_read_ahead()), a single recv() pulls in up to len bytes:
int	**wolfSSL_set_default_read_buffer_len**(WOLFSSL * ssl, size_t len)This function sets the read_ahead coalescing buffer size on a single WOLFSSL session, overriding the value inherited from its WOLFSSL_CTX. See wolfSSL_CTX_set_default_read_buffer_len() for the full description, the one-record default, and the memory-vs-syscall trade-off.
long	**wolfSSL_CTX_get_default_read_buffer_len**(WOLFSSL_CTX * ctx)This function returns the read_ahead coalescing buffer size configured on a WOLFSSL_CTX by wolfSSL_CTX_set_default_read_buffer_len(). Because a length of 0 and oversized values are normalised when set, the value reported is the effective window actually in use (the one_record default rather than 0 when the length was never changed), not the raw argument last passed.

	Name
long	**wolfSSL_get_default_read_buffer_len**(const WOLFSSL * ssl)This function returns the read_ahead coalescing buffer size in effect for a single WOLFSSL session, whether inherited from its WOLFSSL_CTX or overridden by wolfSSL_set_default_read_buffer_len(). As with the WOLFSSL_CTX getter, the value reported is the effective window in use, not the raw argument last passed.
int	**wolfSSL_CTX_set_max_send_fragment**(WOLFSSL_CTX * ctx, long max_fragment)This function caps the plaintext carried in each record sent by WOLFSSL objects created from the context afterwards. It is the wolfSSL equivalent of OpenSSL's SSL_CTX_set_max_send_fragment(). The cap is local and send only: the peer is not told and records received are not limited. Unset, there is no cap. wolfSSL_new() copies the value, so a later call does not change existing objects, as in OpenSSL. With a negotiated max_fragment_length the smaller of the two limits applies, where OpenSSL uses the negotiated length instead. A DTLS write larger than the cap fails with DTLS_SIZE_ERROR, like one larger than the MTU, unless WOLFSSL_NO_DTLS_SIZE_CHECK is defined. In TLS 1.3 only Certificate and CertificateVerify are split to the cap. Other handshake messages, such as ClientHello, ServerHello, EncryptedExtensions, CertificateRequest and NewSessionTicket, are sent in one record each, as they are with a negotiated max_fragment_length. Application data is always split.
int	**wolfSSL_set_max_send_fragment**(WOLFSSL * ssl, long max_fragment)This function caps the plaintext carried in each record sent by a single WOLFSSL object, replacing the value copied from its WOLFSSL_CTX. It is the wolfSSL equivalent of OpenSSL's SSL_set_max_send_fragment() and takes effect from the next record sent. See wolfSSL_CTX_set_max_send_fragment() for the full description.
long	**wolfSSL_CTX_set_tlsext_status_arg**(WOLFSSL_CTX * ctx, void * arg)This function sets the argument to be passed to the OCSP status callback.
void	**wolfSSL_CTX_set_client_cert_cb**(WOLFSSL_CTX * ctx, client_cert_cb cb)Sets a callback to select the client certificate and private key.

	Name
void	**wolfSSL_CTX_set_cert_cb**(WOLFSSL_CTX * ctx, CertSetupCallback cb, void * arg)Sets a generic certificate setup callback.
int	**wolfSSL_CTX_set_tlsext_status_cb**(WOLFSSL_CTX * ctx, tlsextStatusCb cb)Sets the callback to be used for handling OCSP status requests (OCSP stapling).
int	**wolfSSL_CTX_get_tlsext_status_cb**(WOLFSSL_CTX * ctx, tlsextStatusCb * cb)Gets the currently set OCSP status callback for the context.
long	**wolfSSL_get_tlsext_status_ocsp_resp**(WOLFSSL * ssl, unsigned char ** resp)Gets the OCSP response that will be sent (stapled) to the peer.
long	**wolfSSL_set_tlsext_status_ocsp_resp**(WOLFSSL * ssl, unsigned char * resp, int len)Sets the OCSP response to be sent (stapled) to the peer.
int	**wolfSSL_set_tlsext_status_ocsp_resp_multi**(WOLFSSL * ssl, unsigned char * resp, int len, word32 idx)Sets multiple OCSP responses for TLS multi-certificate chains.
void	**wolfSSL_CTX_set_ocsp_status_verify_cb**(WOLFSSL_CTX * ctx, ocspVerifyStatusCb cb, void * cbArg)Sets a callback to verify the OCSP status response.
long	**wolfSSL_CTX_set_tlsext_opaque_prf_input_callback_arg**(WOLF * ctx, void * arg)This function sets the optional argument to be passed to the PRF callback.
long	**wolfSSL_CTX_set_options**(WOLFSSL_CTX * ctx, long opt)This function ORs the bits in opt into the options mask of the given WOLFSSL_CTX. The options mask is inherited by every WOLFSSL session later created from this context. Bits are accumulated — to remove an option, use wolfSSL_CTX_clear_options(). The OpenSSL-style "SSL_OP_" *macros are aliases for the corresponding "WOLFSSL_OP_"* values; either spelling may be used.
long	**wolfSSL_set_options**(WOLFSSL * s, long op)This function ORs the bits in op into the options mask of the given WOLFSSL session. The set of recognized "SSL_OP_" / *"WOLFSSL_OP_"* flags is identical to that documented for wolfSSL_CTX_set_options(); refer to that function for the full options table and the build-option requirements of individual flags. Flags inherited from the parent WOLFSSL_CTX remain set; setting SSL_OP_NO_TLSv1_3 here lowers the session's negotiated minor version to TLS 1.2.
long	**wolfSSL_get_options**(const WOLFSSL * s)This function returns the current options mask.

	Name
long	**wolfSSL_set_tlsext_debug_arg**(WOLFSSL * s, void * arg)This is used to set the debug argument passed around.
long	**wolfSSL_set_tlsext_debug_callback**(WOLFSSL * s, WOLFSSL_TLSEXT_DEBUG_CB cb)This is used to set the TLS extension debug callback on the object.
long	**wolfSSL_set_tlsext_status_type**(WOLFSSL * s, int type)This function is called when the client application request that a server send back an OCSP status response (also known as OCSP stapling).Currently, the only supported type is TLSEXT_STATUSTYPE_ocsp.
long	**wolfSSL_get_verify_result**(const WOLFSSL * ssl)This is used to get the results after trying to verify the peer's certificate.
void	**wolfSSL_ERR_print_errors_fp**(XFILE fp, int err)This function converts an error code returned by wolfSSL_get_error() into a more human_readable error string and prints that string to the output file _ fp. err is the error code returned by wolfSSL_get_error() and fp is the file which the error string will be placed in.
void	**wolfSSL_ERR_print_errors_cb**(int*)(const char* str, size_t len, void *u) cb, void* u)This function uses the provided callback to handle error reporting. The callback function is executed for each error line. The string, length, and userdata are passed into the callback parameters.
void	**wolfSSL_CTX_set_psk_client_callback**(WOLFSSL_CTX * ctx, wc_psk_client_callback cb)The function sets the client_psk_cb member of the WOLFSSL_CTX structure.
void	**wolfSSL_set_psk_client_callback**(WOLFSSL * ssl, wc_psk_client_callback cb)Sets the PSK client side callback.
const char *	**wolfSSL_get_psk_identity_hint**(const WOLFSSL *)This function returns the psk identity hint.
const char *	**wolfSSL_get_psk_identity**(const WOLFSSL *)The function returns a constant pointer to the client_identity member of the Arrays structure.
int	**wolfSSL_CTX_use_psk_identity_hint**(WOLFSSL_CTX * ctx, const char * hint)This function stores the hint argument in the server_hint member of the WOLFSSL_CTX structure.
int	**wolfSSL_use_psk_identity_hint**(WOLFSSL * ssl, const char * hint)This function stores the hint argument in the server_hint member of the Arrays structure within the WOLFSSL structure.

	Name
void	**wolfSSL_CTX_set_psk_server_callback**(WOLFSSL_CTX * ctx, wc_psk_server_callback cb)This function sets the psk callback for the server side in the WOLFSSL_CTX structure.
void	**wolfSSL_set_psk_server_callback**(WOLFSSL * ssl, wc_psk_server_callback cb)Sets the psk callback for the server side by setting the WOLFSSL structure options members.
int	**wolfSSL_set_psk_callback_ctx**(WOLFSSL * ssl, void * psk_ctx)Sets a PSK user context in the WOLFSSL structure options member.
int	**wolfSSL_CTX_set_psk_callback_ctx**(WOLFSSL_CTX * ctx, void * psk_ctx)Sets a PSK user context in the WOLFSSL_CTX structure.
void *	**wolfSSL_get_psk_callback_ctx**(WOLFSSL * ssl)Get a PSK user context in the WOLFSSL structure options member.
void *	**wolfSSL_CTX_get_psk_callback_ctx**(WOLFSSL_CTX * ctx)Get a PSK user context in the WOLFSSL_CTX structure.
int	**wolfSSL_CTX_allow_anon_cipher**(WOLFSSL_CTX * ctx)This function enables the havAnon member of the CTX structure if HAVE_ANON is defined during compilation.
WOLFSSL_METHOD *	**wolfSSLv23_server_method**(void)The wolfSSLv23_server_method() function is used to indicate that the application is a server and will support clients connecting with protocol version from SSL 3.0 _ TLS 1.3. This function allocates memory for and initializes a new WOLFSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().
int	**wolfSSL_state**(WOLFSSL * ssl)This is used to get the internal error state of the WOLFSSL structure.
WOLFSSL_X509 *	**wolfSSL_get_peer_certificate**(WOLFSSL * ssl)This function gets the peer's certificate.
int	**wolfSSL_want_read**(WOLFSSL * ssl)This function is similar to calling wolfSSL_get_error() and getting SSL_ERROR_WANT_READ in return. If the underlying error state is SSL_ERROR_WANT_READ, this function will return 1, otherwise, 0.
int	**wolfSSL_want_write**(WOLFSSL * ssl)This function is similar to calling wolfSSL_get_error() and getting SSL_ERROR_WANT_WRITE in return. If the underlying error state is SSL_ERROR_WANT_WRITE, this function will return 1, otherwise, 0.

	Name
int	**wolfSSL_want**(WOLFSSL * ssl)This function reports which I/O operation, if any, the SSL session is waiting on. It reflects the same state that wolfSSL_want_read() and wolfSSL_want_write() report individually. Unlike those two, which are always available, this function is only built when OPENSSL_EXTRA is defined.
int	**wolfSSL_check_domain_name**(WOLFSSL * ssl, const char * dn)wolfSSL by default checks the peer certificate for a valid date range and a verified signature. Calling this function before wolfSSL_connect() or wolfSSL_accept() will add a domain name check to the list of checks to perform. dn holds the domain name to check against the peer certificate when it's received.
int	**wolfSSL_check_ip_address**(WOLFSSL * ssl, const char * ipaddr)Calling this function before wolfSSL_connect() or wolfSSL_accept() adds an IP-address identity check against the peer certificate SAN iPAddress entries.
int	**wolfSSL_Init**(void)Initializes the wolfSSL library for use. Must be called once per application and before any other call to the library.
int	**wolfSSL_Cleanup**(void)Un-initializes the wolfSSL library from further use. Doesn't have to be called, though it will free any resources used by the library.
const char *	**wolfSSL_lib_version**(void)This function returns the current library version.
word32	**wolfSSL_lib_version_hex**(void)This function returns the current library version in hexadecimal notation.
int	**wolfSSL_negotiate**(WOLFSSL * ssl)Performs the actual connect or accept based on the side of the SSL method. If called from the client side then an wolfSSL_connect() is done while a wolfSSL_accept() is performed if called from the server side.

COPYRIGHT ©2024 wolfSSL Inc.

	Name
int	**wolfSSL_set_compression**(WOLFSSL * ssl)Turns on the ability to use compression for the SSL connection. Both sides must have compression turned on otherwise compression will not be used. The zlib library performs the actual data compression. To compile into the library use –with-libz for the configure system and define HAVE_LIBZ otherwise. Keep in mind that while compressing data before sending decreases the actual size of the messages being sent and received, the amount of data saved by compression usually takes longer in time to analyze than it does to send it raw on all but the slowest of networks.
int	**wolfSSL_set_timeout**(WOLFSSL * ssl, unsigned int to)This function sets the SSL session timeout value in seconds.
int	**wolfSSL_CTX_set_timeout**(WOLFSSL_CTX * ctx, unsigned int to)This function sets the timeout value for SSL sessions, in seconds, for the specified SSL context.
WOLFSSL_X509_CHAIN *	**wolfSSL_get_peer_chain**(WOLFSSL * ssl)Retrieves the peer's certificate chain.
int	**wolfSSL_get_chain_count**(WOLFSSL_X509_CHAIN * chain)Retrieve's the peers certificate chain count.
int	**wolfSSL_get_chain_length**(WOLFSSL_X509_CHAIN * chain, int idx)Retrieves the peer's ASN1.DER certificate length in bytes at index (idx).
unsigned char *	**wolfSSL_get_chain_cert**(WOLFSSL_X509_CHAIN * chain, int idx)Retrieves the peer's ASN1.DER certificate at index (idx).
WOLFSSL_X509 *	**wolfSSL_get_chain_X509**(WOLFSSL_X509_CHAIN * chain, int idx)This function gets the peer's wolfSSL_X509_certificate at index (idx) from the chain of certificates.
int	**wolfSSL_get_chain_cert_pem**(WOLFSSL_X509_CHAIN * chain, int idx, unsigned char * buf, int inLen, int * outLen)Retrieves the peer's PEM certificate at index (idx).
const unsigned char *	**wolfSSL_get_sessionID**(const WOLFSSL_SESSION * s)Retrieves the session's ID. The session ID is always 32 bytes long.
int	**wolfSSL_X509_get_serial_number**(WOLFSSL_X509 * x509, unsigned char * in, int * inOutSz)Retrieves the peer's certificate serial number. The serial number buffer (in) should be at least 32 bytes long and be provided as the *inOutSz argument as input. After calling the function* inOutSz will hold the actual length in bytes written to the in buffer.

	Name
char *	**wolfSSL_X509_get_subjectCN**(WOLFSSL_X509 *)Returns the common name of the subject from the certificate.
const unsigned char *	**wolfSSL_X509_get_der**(WOLFSSL_X509 * x509, int * outSz)This function gets the DER encoded certificate in the WOLFSSL_X509 struct.
WOLFSSL_ASN1_TIME *	**wolfSSL_X509_get_notAfter**(WOLFSSL_X509 *)This function checks to see if x509 is NULL and if it's not, it returns the notAfter member of the x509 struct.
int	**wolfSSL_X509_version**(WOLFSSL_X509 * x509)This function retrieves the version of the X509 certificate.
WOLFSSL_X509 *	**wolfSSL_X509_d2i_fp**(WOLFSSL_X509 ** x509, FILE * file)If NO_STDIO_FILESYSTEM is defined this function will allocate heap memory, initialize a WOLFSSL_X509 structure and return a pointer to it.
WOLFSSL_X509 *	**wolfSSL_X509_load_certificate_file**(const char * fname, int format)The function loads the x509 certificate into memory.
unsigned char *	**wolfSSL_X509_get_device_type**(WOLFSSL_X509 * x509, unsigned char * in, int * inOutSz)This function copies the device type from the x509 structure to the buffer.
unsigned char *	**wolfSSL_X509_get_hw_type**(WOLFSSL_X509 * x509, unsigned char * in, int * inOutSz)The function copies the hwType member of the WOLFSSL_X509 structure to the buffer.
unsigned char *	**wolfSSL_X509_get_hw_serial_number**(WOLFSSL_X509 * x509, unsigned char * in, int * inOutSz)This function returns the hwSerialNum member of the x509 object.

COPYRIGHT ©2024 wolfSSL Inc.

	Name
int	**wolfSSL_connect_cert**(WOLFSSL * ssl)This function is called on the client side and initiates an SSL/TLS handshake with a server only long enough to get the peer's certificate chain. When this function is called, the underlying communication channel has already been set up. wolfSSL_connect_cert() works with both blocking and non_blocking I/O. When the underlying I/O is non_blocking, wolfSSL_connect_cert() will return when the underlying I/O could not satisfy the needs of wolfSSL_connect_cert() to continue the handshake. In this case, a call to wolfSSL_get_error() will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_connect_cert() when the underlying I/O is ready and wolfSSL will pick up where it left off. When using a non_blocking socket, nothing needs to be done, but select() can be used to check for the required condition. If the underlying I/O is blocking, wolfSSL_connect_cert() will only return once the peer's certificate chain has been received.
WC_PKCS12 *	**wolfSSL_d2i_PKCS12_bio**(WOLFSSL_BIO * bio, WC_PKCS12 ** pkcs12)wolfSSL_d2i_PKCS12_bio (d2i_PKCS12_bio) copies in the PKCS12 information from WOLFSSL_BIO to the structure WC_PKCS12. The information is divided up in the structure as a list of Content Infos along with a structure to hold optional MAC information. After the information has been divided into chunks (but not decrypted) in the structure WC_PKCS12, it can then be parsed and decrypted by calling.
WC_PKCS12 *	**wolfSSL_i2d_PKCS12_bio**(WOLFSSL_BIO * bio, WC_PKCS12 * pkcs12)wolfSSL_i2d_PKCS12_bio (i2d_PKCS12_bio) copies in the cert information from the structure WC_PKCS12 to WOLFSSL_BIO.

	Name
int	**wolfSSL_PKCS12_parse**(WC_PKCS12 * pkcs12, const char * psw, WOLFSSL_EVP_PKEY ** pkey, WOLFSSL_X509 ** cert, WOLF_STACK_OF(WOLFSSL_X509) ** ca)PKCS12 can be enabled with adding –enable_opensslextra to the configure command. It can use triple DES and RC4 for decryption so would recommend also enabling these features when enabling opensslextra (–enable_des3 –enable_arc4). wolfSSL does not currently support RC2 so decryption with RC2 is currently not available. This may be noticeable with default encryption schemes used by OpenSSL command line to create .p12 files. wolfSSL_PKCS12_parse (PKCS12_parse). The first thing this function does is check the MAC is correct if present. If the MAC fails then the function returns and does not try to decrypt any of the stored Content Infos. This function then parses through each Content Info looking for a bag type, if the bag type is known it is decrypted as needed and either stored in the list of certificates being built or as a key found. After parsing through all bags the key found is then compared with the certificate list until a matching pair is found. This matching pair is then returned as the key and certificate, optionally the certificate list found is returned as a STACK_OF certificates. At the moment a CRL, Secret or SafeContents bag will be skipped over and not parsed. It can be seen if these or other “Unknown” bags are skipped over by viewing the debug print out. Additional attributes such as friendly name are skipped over when parsing a PKCS12 file.
int	**wolfSSL_SetTmpDH**(WOLFSSL * ssl, const unsigned char * p, int pSz, const unsigned char * g, int gSz)Server Diffie-Hellman Ephemeral parameters setting. This function sets up the group parameters to be used if the server negotiates a cipher suite that uses DHE.
int	**wolfSSL_SetTmpDH_buffer**(WOLFSSL * ssl, const unsigned char * b, long sz, int format)The function calls the wolfSSL_SetTMpDH_buffer_wrapper, which is a wrapper for Diffie-Hellman parameters.
int	**wolfSSL_SetTmpDH_file**(WOLFSSL * ssl, const char * f, int format)This function calls wolfSSL_SetTmpDH_file_wrapper to set server Diffie-Hellman parameters.

	Name
int	**wolfSSL_CTX_SetTmpDH**(WOLFSSL_CTX * ctx, const unsigned char * p, int pSz, const unsigned char * g, int gSz)Sets the parameters for the server CTX Diffie-Hellman.
int	**wolfSSL_CTX_SetTmpDH_buffer**(WOLFSSL_CTX * ctx, const unsigned char * b, long sz, int format)A wrapper function that calls wolfSSL_SetTmpDH_buffer_wrapper.
int	**wolfSSL_CTX_SetTmpDH_file**(WOLFSSL_CTX * ctx, const char * f, int format)The function calls wolfSSL_SetTmpDH_file_wrapper to set the server Diffie-Hellman parameters.
int	**wolfSSL_CTX_SetMinDhKey_Sz**(WOLFSSL_CTX * ctx, word16 keySz_bits)This function sets the minimum size (in bits) of the Diffie Hellman key size by accessing the minDhKeySz member in the WOLFSSL_CTX structure.
int	**wolfSSL_SetMinDhKey_Sz**(WOLFSSL * ssl, word16 keySz_bits)Sets the minimum size (in bits) for a Diffie-Hellman key in the WOLFSSL structure.
int	**wolfSSL_CTX_SetMaxDhKey_Sz**(WOLFSSL_CTX * ctx, word16 keySz_bits)This function sets the maximum size (in bits) of the Diffie Hellman key size by accessing the maxDhKeySz member in the WOLFSSL_CTX structure.
int	**wolfSSL_SetMaxDhKey_Sz**(WOLFSSL * ssl, word16 keySz_bits)Sets the maximum size (in bits) for a Diffie-Hellman key in the WOLFSSL structure.
int	**wolfSSL_GetDhKey_Sz**(WOLFSSL * ssl)Returns the value of dhKeySz (in bits) that is a member of the options structure. This value represents the Diffie-Hellman key size in bytes.
int	**wolfSSL_CTX_SetMinRsaKey_Sz**(WOLFSSL_CTX * ctx, short keySz)Sets the minimum RSA key size in both the WOLFSSL_CTX structure and the WOLFSSL_CERT_MANAGER structure.
int	**wolfSSL_SetMinRsaKey_Sz**(WOLFSSL * ssl, short keySz)Sets the minimum allowable key size in bits for RSA located in the WOLFSSL structure.
int	**wolfSSL_CTX_SetMinEccKey_Sz**(WOLFSSL_CTX * ctx, short keySz)Sets the minimum size in bits for the ECC key in the WOLF_CTX structure and the WOLFSSL_CERT_MANAGER structure.
int	**wolfSSL_SetMinEccKey_Sz**(WOLFSSL * ssl, short keySz)Sets the value of the minEccKeySz member of the options structure. The options struct is a member of the WOLFSSL structure and is accessed through the ssl parameter.

	Name
int	**wolfSSL_make_eap_keys**(WOLFSSL * ssl, void * key, unsigned int len, const char * label)This function is used by EAP_TLS and EAP-TTLS to derive keying material from the master secret.
int	**wolfSSL_writev**(WOLFSSL * ssl, const struct iovec * iov, int iovcnt)Simulates writev semantics but doesn't actually do block at a time because of SSL_write() behavior and because front adds may be small. Makes porting into software that uses writev easier.
int	**wolfSSL_CTX_UnloadCAs**(WOLFSSL_CTX * ctx)This function unloads the CA signer list and frees the whole signer table.
int	**wolfSSL_CTX_UnloadIntermediateCerts**(WOLFSSL_CTX * ctx)This function unloads intermediate certificates added to the CA signer list and frees them.
int	**wolfSSL_CTX_Unload_trust_peers**(WOLFSSL_CTX * ctx)This function is used to unload all previously loaded trusted peer certificates. Feature is enabled by defining the macro WOLFSSL_TRUST_PEER_CERT.
int	**wolfSSL_CTX_trust_peer_buffer**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format)This function loads a certificate to use for verifying a peer when performing a TLS/SSL handshake. The peer certificate sent during the handshake is compared by using the SKID when available and the signature. If these two things do not match then any loaded CAs are used. Is the same functionality as wolfSSL_CTX_trust_peer_cert except is from a buffer instead of a file. Feature is enabled by defining the macro WOLFSSL_TRUST_PEER_CERT Please see the examples for proper usage.
int	**wolfSSL_CTX_load_verify_buffer**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format)This function loads a CA certificate buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. More than one CA certificate may be loaded per buffer as long as the format is in PEM. Please see the examples for proper usage.

	Name
int	**wolfSSL_CTX_load_verify_buffer_ex**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format, int userChain, word32 flags)This function loads a CA certificate buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. More than one CA certificate may be loaded per buffer as long as the format is in PEM. The _ex version was added in PR 2413 and supports additional arguments for userChain and flags.
int	**wolfSSL_CTX_load_verify_chain_buffer_format**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format)This function loads a CA certificate chain buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. More than one CA certificate may be loaded per buffer as long as the format is in PEM. Please see the examples for proper usage.
int	**wolfSSL_CTX_use_certificate_buffer**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format)This function loads a certificate buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. Please see the examples for proper usage.
int	**wolfSSL_CTX_use_PrivateKey_buffer**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format)This function loads a private key buffer into the SSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1or SSL_FILETYPE_PEM. Please see the examples for proper usage.

	Name
int	**wolfSSL_CTX_use_certificate_chain_buffer**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz)This function loads a certificate chain buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. The buffer must be in PEM format and start with the subject's certificate, ending with the root certificate. Please see the examples for proper usage.
int	**wolfSSL_use_certificate_buffer**(WOLFSSL * ssl, const unsigned char * in, long sz, int format)This function loads a certificate buffer into the WOLFSSL object. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. Please see the examples for proper usage.
int	**wolfSSL_use_PrivateKey_buffer**(WOLFSSL * ssl, const unsigned char * in, long sz, int format)This function loads a private key buffer into the WOLFSSL object. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. Please see the examples for proper usage.
int	**wolfSSL_use_certificate_chain_buffer**(WOLFSSL * ssl, const unsigned char * in, long sz)This function loads a certificate chain buffer into the WOLFSSL object. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. The buffer must be in PEM format and start with the subject's certificate, ending with the root certificate. Please see the examples for proper usage.
int	**wolfSSL_UnloadCertsKeys**(WOLFSSL * ssl)This function unloads any certificates or keys that SSL owns.
int	**wolfSSL_CTX_set_group_messages**(WOLFSSL_CTX * ctx)This function turns on grouping of handshake messages where possible.

	Name
int	**wolfSSL_set_group_messages**(WOLFSSL * ssl)This function turns on grouping of handshake messages where possible.
void	**wolfSSL_SetFuzzerCb**(WOLFSSL * ssl, CallbackFuzzer cbf, void * fCtx)This function sets the fuzzer callback.
int	**wolfSSL_DTLS_SetCookieSecret**(WOLFSSL * ssl, const byte * secret, word32 secretSz)This function sets a new dtls cookie secret.
int	**wolfSSL_DTLS_SetCookieSecretSecondary**(WOLFSSL * ssl, const byte * secret, word32 secretSz)This function sets a secondary DTLS 1.2 cookie secret used only when verifying a received HelloVerifyRequest cookie, and only if the primary secret (set with wolfSSL_DTLS_SetCookieSecret()) fails to verify it. This lets an application rotate the cookie secret on a stateless server without rejecting clients whose cookie was issued under the previous secret: install the new secret as the primary and the previous secret here for an overlap window. The secondary secret is never used to issue cookies. It is the DTLS 1.2 counterpart of wolfSSL_set_hrr_cookie_secret_secondary().
WC_RNG *	**wolfSSL_GetRNG**(WOLFSSL * ssl)This function retrieves the random number.
int	**wolfSSL_CTX_SetMinVersion**(WOLFSSL_CTX * ctx, int version)This function sets the minimum downgrade version allowed. Applicable only when the connection allows downgrade using (wolfSSLv23_client_method or wolfSSLv23_server_method).
int	**wolfSSL_SetMinVersion**(WOLFSSL * ssl, int version)This function sets the minimum downgrade version allowed. Applicable only when the connection allows downgrade using (wolfSSLv23_client_method or wolfSSLv23_server_method).
int	**wolfSSL_GetObjectSize**(void)This function returns the size of the WOLFSSL object and will be dependent on build options and settings. If SHOW_SIZES has been defined when building wolfSSL, this function will also print the sizes of individual objects within the WOLFSSL object (Suites, Ciphers, etc.) to stdout.

COPYRIGHT ©2024 wolfSSL Inc.

	Name
int	**wolfSSL_GetOutputSize**(WOLFSSL * ssl, int inSz)Returns the record layer size of the plaintext input. This is helpful when an application wants to know how many bytes will be sent across the Transport layer, given a specified plaintext input size. This function must be called after the SSL/TLS handshake has been completed.
int	**wolfSSL_GetMaxOutputSize**(WOLFSSL * ssl)Returns the maximum record layer size for plaintext data. This will correspond to either the maximum SSL/TLS record size as specified by the protocol standard, the maximum TLS fragment size as set by the TLS Max Fragment Length extension. This function is helpful when the application has called wolfSSL_GetOutputSize() and received a INPUT_SIZE_E error. This function must be called after the SSL/TLS handshake has been completed.
int	**wolfSSL_SetVersion**(WOLFSSL * ssl, int version)This function sets the SSL/TLS protocol version for the specified SSL session (WOLFSSL object) using the version as specified by version. This will override the protocol setting for the SSL session (ssl) _ originally defined and set by the SSL context (wolfSSL_CTX_new()) method type.
void	**wolfSSL_CTX_SetMacEncryptCb**(WOLFSSL_CTX * ctx, CallbackMacEncrypt cb)Allows caller to set the Atomic User Record Processing Mac/Encrypt Callback. The callback should return 0 for success or < 0 for an error. The ssl and ctx pointers are available for the user's convenience. macOut is the output buffer where the result of the mac should be stored. macIn is the mac input buffer and macInSz notes the size of the buffer. macContent and macVerify are needed for wolfSSL_SetTlsHmacInner() and be passed along as is. encOut is the output buffer where the result on the encryption should be stored. encIn is the input buffer to encrypt while encSz is the size of the input. An example callback can be found wolfssl/test.h myMacEncryptCb().
void	**wolfSSL_SetMacEncryptCtx**(WOLFSSL * ssl, void * ctx)Allows caller to set the Atomic User Record Processing Mac/Encrypt Callback Context to ctx.

	Name
void *	**wolfSSL_GetMacEncryptCtx**(WOLFSSL * ssl)Allows caller to retrieve the Atomic User Record Processing Mac/Encrypt Callback Context previously stored with wolfSSL_SetMacEncryptCtx().
void	**wolfSSL_CTX_SetDecryptVerifyCb**(WOLFSSL_CTX * ctx, CallbackDecryptVerify cb)Allows caller to set the Atomic User Record Processing Decrypt/Verify Callback. The callback should return 0 for success or < 0 for an error. The ssl and ctx pointers are available for the user's convenience. decOut is the output buffer where the result of the decryption should be stored. decIn is the encrypted input buffer and decInSz notes the size of the buffer. content and verify are needed for wolfSSL_SetTlsHmacInner() and be passed along as is. padSz is an output variable that should be set with the total value of the padding. That is, the mac size plus any padding and pad bytes. An example callback can be found wolfssl/test.h myDecryptVerifyCb().
void	**wolfSSL_SetDecryptVerifyCtx**(WOLFSSL * ssl, void * ctx)Allows caller to set the Atomic User Record Processing Decrypt/Verify Callback Context to ctx.
void *	**wolfSSL_GetDecryptVerifyCtx**(WOLFSSL * ssl)Allows caller to retrieve the Atomic User Record Processing Decrypt/Verify Callback Context previously stored with wolfSSL_SetDecryptVerifyCtx().
const unsigned char *	**wolfSSL_GetMacSecret**(WOLFSSL * ssl, int verify)Allows retrieval of the Hmac/Mac secret from the handshake process. The verify parameter specifies whether this is for verification of a peer message.
const unsigned char *	**wolfSSL_GetClientWriteKey**(WOLFSSL *)Allows retrieval of the client write key from the handshake process.
const unsigned char *	**wolfSSL_GetClientWriteIV**(WOLFSSL *)Allows retrieval of the client write IV (initialization vector) from the handshake process.
const unsigned char *	**wolfSSL_GetServerWriteKey**(WOLFSSL *)Allows retrieval of the server write key from the handshake process.
const unsigned char *	**wolfSSL_GetServerWriteIV**(WOLFSSL *)Allows retrieval of the server write IV (initialization vector) from the handshake process.
int	**wolfSSL_GetKeySize**(WOLFSSL * ssl)Allows retrieval of the key size from the handshake process.

	Name
int	**wolfSSL_GetIVSize**(WOLFSSL * ssl)Returns the iv_size member of the specs structure held in the WOLFSSL struct.
int	**wolfSSL_GetSide**(WOLFSSL * ssl)Allows retrieval of the side of this WOLFSSL connection.
int	**wolfSSL_IsTLSv1_1**(WOLFSSL * ssl)Allows caller to determine if the negotiated protocol version is at least TLS version 1.1 or greater.
int	**wolfSSL_GetBulkCipher**(WOLFSSL * ssl)Allows caller to determine the negotiated bulk cipher algorithm from the handshake.
int	**wolfSSL_GetCipherBlockSize**(WOLFSSL * ssl)Allows caller to determine the negotiated cipher block size from the handshake.
int	**wolfSSL_GetAeadMacSize**(WOLFSSL * ssl)Allows caller to determine the negotiated aead mac size from the handshake. For cipher type WOLFSSL_AEAD_TYPE.
int	**wolfSSL_GetHmacSize**(WOLFSSL * ssl)Allows caller to determine the negotiated (h)mac size from the handshake. For cipher types except WOLFSSL_AEAD_TYPE.
int	**wolfSSL_GetHmacType**(WOLFSSL * ssl)Allows caller to determine the negotiated (h)mac type from the handshake. For cipher types except WOLFSSL_AEAD_TYPE.
int	**wolfSSL_GetCipherType**(WOLFSSL * ssl)Allows caller to determine the negotiated cipher type from the handshake.
int	**wolfSSL_SetTlsHmacInner**(WOLFSSL * ssl, byte * inner, word32 sz, int content, int verify)Allows caller to set the Hmac Inner vector for message sending/receiving. The result is written to inner which should be at least wolfSSL_GetHmacSize() bytes. The size of the message is specified by sz, content is the type of message, and verify specifies whether this is a verification of a peer message. Valid for cipher types excluding WOLFSSL_AEAD_TYPE.

	Name
void	**wolfSSL_CTX_SetEccSignCb**(WOLFSSL_CTX * ctx, CallbackEccSign cb)Allows caller to set the Public Key Callback for ECC Signing. The callback should return 0 for success or < 0 for an error. The ssl and ctx pointers are available for the user's convenience. in is the input buffer to sign while inSz denotes the length of the input. out is the output buffer where the result of the signature should be stored. outSz is an input/output variable that specifies the size of the output buffer upon invocation and the actual size of the signature should be stored there before returning. keyDer is the ECC Private key in ASN1 format and keySz is the length of the key in bytes. An example callback can be found wolfssl/test.h myEccSign().
void	**wolfSSL_SetEccSignCtx**(WOLFSSL * ssl, void * ctx)Allows caller to set the Public Key Ecc Signing Callback Context to ctx.
void *	**wolfSSL_GetEccSignCtx**(WOLFSSL * ssl)Allows caller to retrieve the Public Key Ecc Signing Callback Context previously stored with wolfSSL_SetEccSignCtx().
void	**wolfSSL_CTX_SetEccSignCtx**(WOLFSSL_CTX * ctx, void * userCtx)Allows caller to set the Public Key Ecc Signing Callback Context to ctx.
void *	**wolfSSL_CTX_GetEccSignCtx**(WOLFSSL_CTX * ctx)Allows caller to retrieve the Public Key Ecc Signing Callback Context previously stored with wolfSSL_SetEccSignCtx().
void	**wolfSSL_CTX_SetEccVerifyCb**(WOLFSSL_CTX * ctx, CallbackEccVerify cb)Allows caller to set the Public Key Callback for ECC Verification. The callback should return 0 for success or < 0 for an error. The ssl and ctx pointers are available for the user's convenience. sig is the signature to verify and sigSz denotes the length of the signature. hash is an input buffer containing the digest of the message and hashSz denotes the length in bytes of the hash. result is an output variable where the result of the verification should be stored, 1 for success and 0 for failure. keyDer is the ECC Private key in ASN1 format and keySz is the length of the key in bytes. keyDer is NULL with keySz 0 when only the callback holds the key, so the callback must then supply it. An example callback can be found wolfssl/test.h myEccVerify().
void	**wolfSSL_SetEccVerifyCtx**(WOLFSSL * ssl, void * ctx)Allows caller to set the Public Key Ecc Verification Callback Context to ctx.

	Name
void *	**wolfSSL_GetEccVerifyCtx**(WOLFSSL * ssl)Allows caller to retrieve the Public Key Ecc Verification Callback Context previously stored with wolfSSL_SetEccVerifyCtx().
void	**wolfSSL_CTX_SetRsaSignCb**(WOLFSSL_CTX * ctx, CallbackRsaSign cb)Allows caller to set the Public Key Callback for RSA Signing. The callback should return 0 for success or < 0 for an error. The ssl and ctx pointers are available for the user's convenience. in is the input buffer to sign while inSz denotes the length of the input. out is the output buffer where the result of the signature should be stored. outSz is an input/output variable that specifies the size of the output buffer upon invocation and the actual size of the signature should be stored there before returning. keyDer is the RSA Private key in ASN1 format and keySz is the length of the key in bytes. An example callback can be found wolfssl/test.h myRsaSign().
void	**wolfSSL_SetRsaSignCtx**(WOLFSSL * ssl, void * ctx)Allows caller to set the Public Key RSA Signing Callback Context to ctx.
void *	**wolfSSL_GetRsaSignCtx**(WOLFSSL * ssl)Allows caller to retrieve the Public Key RSA Signing Callback Context previously stored with wolfSSL_SetRsaSignCtx().
void	**wolfSSL_CTX_SetRsaVerifyCb**(WOLFSSL_CTX * ctx, CallbackRsaVerify cb)Allows caller to set the Public Key Callback for RSA Verification. The callback should return the number of plaintext bytes for success or < 0 for an error. The ssl and ctx pointers are available for the user's convenience. sig is the signature to verify and sigSz denotes the length of the signature. out should be set to the beginning of the verification buffer after the decryption process and any padding. keyDer is the RSA Public key in ASN1 format and keySz is the length of the key in bytes. An example callback can be found wolfssl/test.h myRsaVerify().
void	**wolfSSL_SetRsaVerifyCtx**(WOLFSSL * ssl, void * ctx)Allows caller to set the Public Key RSA Verification Callback Context to ctx.
void *	**wolfSSL_GetRsaVerifyCtx**(WOLFSSL * ssl)Allows caller to retrieve the Public Key RSA Verification Callback Context previously stored with wolfSSL_SetRsaVerifyCtx().

	Name
void	**wolfSSL_CTX_SetRsaEncCb**(WOLFSSL_CTX * ctx, CallbackRsaEnc cb)Allows caller to set the Public Key Callback for RSA Public Encrypt. The callback should return 0 for success or < 0 for an error. The ssl and ctx pointers are available for the user's convenience. in is the input buffer to encrypt while inSz denotes the length of the input. out is the output buffer where the result of the encryption should be stored. outSz is an input/output variable that specifies the size of the output buffer upon invocation and the actual size of the encryption should be stored there before returning. keyDer is the RSA Public key in ASN1 format and keySz is the length of the key in bytes. An example callback can be found wolfssl/test.h myRsaEnc().
void	**wolfSSL_SetRsaEncCtx**(WOLFSSL * ssl, void * ctx)Allows caller to set the Public Key RSA Public Encrypt Callback Context to ctx.
void *	**wolfSSL_GetRsaEncCtx**(WOLFSSL * ssl)Allows caller to retrieve the Public Key RSA Public Encrypt Callback Context previously stored with wolfSSL_SetRsaEncCtx().
void	**wolfSSL_CTX_SetRsaDecCb**(WOLFSSL_CTX * ctx, CallbackRsaDec cb)Allows caller to set the Public Key Callback for RSA Private Decrypt. The callback should return the number of plaintext bytes for success or < 0 for an error. The ssl and ctx pointers are available for the user's convenience. in is the input buffer to decrypt and inSz denotes the length of the input. out should be set to the beginning of the decryption buffer after the decryption process and any padding. keyDer is the RSA Private key in ASN1 format and keySz is the length of the key in bytes. An example callback can be found wolfssl/test.h myRsaDec().
void	**wolfSSL_SetRsaDecCtx**(WOLFSSL * ssl, void * ctx)Allows caller to set the Public Key RSA Private Decrypt Callback Context to ctx.
void *	**wolfSSL_GetRsaDecCtx**(WOLFSSL * ssl)Allows caller to retrieve the Public Key RSA Private Decrypt Callback Context previously stored with wolfSSL_SetRsaDecCtx().
void	**wolfSSL_CTX_SetCACb**(WOLFSSL_CTX * ctx, CallbackCACache cb)This function registers a callback with the SSL context (WOLFSSL_CTX) to be called when a new CA certificate is loaded into wolfSSL. The callback is given a buffer with the DER-encoded certificate.

	Name
WOLFSSL_CERT_MANAGER *	**wolfSSL_CertManagerNew_ex**(void * heap)Allocates and initializes a new Certificate Manager context. This context may be used independent of SSL needs. It may be used to load certificates, verify certificates, and check the revocation status.
WOLFSSL_CERT_MANAGER *	**wolfSSL_CertManagerNew**(void)Allocates and initializes a new Certificate Manager context. This context may be used independent of SSL needs. It may be used to load certificates, verify certificates, and check the revocation status.
void	**wolfSSL_CertManagerFree**(WOLFSSL_CERT_MANAGER * cm)Frees all resources associated with the Certificate Manager context. Call this when you no longer need to use the Certificate Manager.
int	**wolfSSL_CertManagerLoadCA**(WOLFSSL_CERT_MANAGER * cm, const char * f, const char * d)Specifies the locations for CA certificate loading into the manager context. The PEM certificate CAfile may contain several trusted CA certificates. If CApath is not NULL it specifies a directory containing CA certificates in PEM format.
int	**wolfSSL_CertManagerLoadCABuffer**(WOLFSSL_CERT_MANAGER * cm, const unsigned char * buff, long sz, int format)Loads the CA Buffer by calling wolfSSL_CTX_load_verify_buffer and returning that result using a temporary cm so as not to lose the information in the cm passed into the function.
int	**wolfSSL_CertManagerUnloadCAs**(WOLFSSL_CERT_MANAGER * cm)This function unloads the CA signer list.
int	**wolfSSL_CertManagerUnloadIntermediateCerts**(WOLFSSL_CE * cm)This function unloads intermediate certificates add to the CA signer list.
int	**wolfSSL_CertManagerUnload_trust_peers**(WOLFSSL_CERT_MA * cm)The function will free the Trusted Peer linked list and unlocks the trusted peer list.
int	**wolfSSL_CertManagerVerify**(WOLFSSL_CERT_MANAGER * cm, const char * f, int format)Specifies the certificate to verify with the Certificate Manager context. The format can be SSL_FILETYPE_PEM or SSL_FILETYPE_ASN1.
int	**wolfSSL_CertManagerVerifyBuffer**(WOLFSSL_CERT_MANAGER * cm, const unsigned char * buff, long sz, int format)Specifies the certificate buffer to verify with the Certificate Manager context. The format can be SSL_FILETYPE_PEM or SSL_FILETYPE_ASN1.

	Name
void	**wolfSSL_CertManagerSetVerify**(WOLFSSL_CERT_MANAGER * cm, VerifyCallback verify_callback)The function sets the verifyCallback function in the Certificate Manager. If present, it will be called for each cert loaded. If there is a verification error, the verify callback can be used to over-ride the error.
int	**wolfSSL_CertManagerCheckCRL**(WOLFSSL_CERT_MANAGER * cm, const unsigned char * der, int sz)Check CRL if the option is enabled and compares the cert to the CRL list.
int	**wolfSSL_CertManagerEnableCRL**(WOLFSSL_CERT_MANAGER * cm, int options)Turns on Certificate Revocation List checking when verifying certificates with the Certificate Manager. By default, CRL checking is off. options include WOLFSSL_CRL_CHECKALL which performs CRL checking on each certificate in the chain versus the Leaf certificate only which is the default.
int	**wolfSSL_CertManagerDisableCRL**(WOLFSSL_CERT_MANAGER * cm)Turns off Certificate Revocation List checking when verifying certificates with the Certificate Manager. By default, CRL checking is off. You can use this function to temporarily or permanently disable CRL checking with this Certificate Manager context that previously had CRL checking enabled.
int	**wolfSSL_CertManagerLoadCRL**(WOLFSSL_CERT_MANAGER * cm, const char * path, int type, int monitor)Error checks and passes through to LoadCRL() in order to load the cert into the CRL for revocation checking. An updated CRL can be loaded by first calling wolfSSL_CertManagerFreeCRL, then loading the new CRL.
int	**wolfSSL_CertManagerLoadCRLBuffer**(WOLFSSL_CERT_MANAGE * cm, const unsigned char * buff, long sz, int type)The function loads the CRL file by calling BufferLoadCRL.
int	**wolfSSL_CertManagerSetCRL_Cb**(WOLFSSL_CERT_MANAGER * cm, CbMissingCRL cb)This function sets the CRL Certificate Manager callback. If HAVE_CRL is defined and a matching CRL record is not found then the cbMissingCRL is called (set via wolfSSL_CertManagerSetCRL_Cb). This allows you to externally retrieve the CRL and load it.

	Name
int	**wolfSSL_CertManagerSetCRLUpdate_Cb**(WOLFSSL_CERT_MANA * cm, CbUpdateCRL cb)This function sets the CRL Update callback. If HAVE_CRL and HAVE_CRL_UPDATE_CB is defined , and an entry with the same issuer and a lower CRL number exists when a CRL is added, then the CbUpdateCRL is called with the details of the existing entry and the new one replacing it.
int	**wolfSSL_CertManagerGetCRLInfo**(WOLFSSL_CERT_MANAGER * cm, CrlInfo * info, const byte * buff, long sz, int type)This function yields a structure with parsed CRL information from an encoded CRL buffer.
int	**wolfSSL_CertManagerCheckOCSP**(WOLFSSL_CERT_MANAGER * cm, const unsigned char * der, int sz)The function enables the WOLFSSL_CERT_MANAGER's member, ocspEnabled to signify that the OCSP check option is enabled.
int	**wolfSSL_CertManagerEnableOCSP**(WOLFSSL_CERT_MANAGER * cm, int options)Turns on OCSP if it's turned off and if compiled with the set option available.
int	**wolfSSL_CertManagerDisableOCSP**(WOLFSSL_CERT_MANAGER * cm)Disables OCSP certificate revocation.
int	**wolfSSL_CertManagerSetOCSPOverrideURL**(WOLFSSL_CERT_M * cm, const char * url)The function copies the url to the ocspOverrideURL member of the WOLFSSL_CERT_MANAGER structure.
int	**wolfSSL_CertManagerSetOCSP_Cb**(WOLFSSL_CERT_MANAGER * cm, CbOCSPIO ioCb, CbOCSPRespFree respFreeCb, void * ioCbCtx)The function sets the OCSP callback in the WOLFSSL_CERT_MANAGER.
int	**wolfSSL_CertManagerEnableOCSPStapling**(WOLFSSL_CERT_MA * cm)This function turns on OCSP stapling if it is not turned on as well as set the options.
int	**wolfSSL_EnableCRL**(WOLFSSL * ssl, int options)Enables CRL certificate revocation.
int	**wolfSSL_DisableCRL**(WOLFSSL * ssl)Disables CRL certificate revocation.
int	**wolfSSL_LoadCRL**(WOLFSSL * ssl, const char * path, int type, int monitor)A wrapper function that ends up calling LoadCRL to load the certificate for revocation checking.
int	**wolfSSL_SetCRL_Cb**(WOLFSSL * ssl, CbMissingCRL cb)Sets the CRL callback in the WOLFSSL_CERT_MANAGER structure.

	Name
int	**wolfSSL_EnableOCSP**(WOLFSSL * ssl, int options)This function enables OCSP certificate verification. The value of options if formed by or'ing one or more of the following options: WOLFSSL_OCSP_URL_OVERRIDE _ use the override URL instead of the URL in certificates. The override URL is specified using the wolfSSL_CTX_SetOCSP_OverrideURL() function. WOLFSSL_OCSP_CHECKALL - Set all OCSP checks on WOLFSSL_OCSP_NO_NONCE - Set nonce option for creating OCSP requests.
int	**wolfSSL_DisableOCSP**(WOLFSSL * ssl)Disables the OCSP certificate revocation option.
int	**wolfSSL_SetOCSP_OverrideURL**(WOLFSSL * ssl, const char * url)This function sets the ocspOverrideURL member in the WOLFSSL_CERT_MANAGER structure.
int	**wolfSSL_SetOCSP_Cb**(WOLFSSL * ssl, CbOCSPIO ioCb, CbOCSPRespFree respFreeCb, void * ioCbCtx)This function sets the OCSP callback in the WOLFSSL_CERT_MANAGER structure.
int	**wolfSSL_CTX_EnableCRL**(WOLFSSL_CTX * ctx, int options)Enables CRL certificate verification through the CTX.
int	**wolfSSL_CTX_DisableCRL**(WOLFSSL_CTX * ctx)This function disables CRL verification in the CTX structure.
int	**wolfSSL_CTX_LoadCRL**(WOLFSSL_CTX * ctx, const char * path, int type, int monitor)This function loads CRL into the WOLFSSL_CTX structure through wolfSSL_CertManagerLoadCRL().
int	**wolfSSL_CTX_SetCRL_Cb**(WOLFSSL_CTX * ctx, CbMissingCRL cb)This function will set the callback argument to the cbMissingCRL member of the WOLFSSL_CERT_MANAGER structure by calling wolfSSL_CertManagerSetCRL_Cb.
int	**wolfSSL_CTX_EnableOCSP**(WOLFSSL_CTX * ctx, int options)This function sets options to configure behavior of OCSP functionality in wolfSSL. The value of options if formed by or'ing one or more of the following options: WOLFSSL_OCSP_URL_OVERRIDE _ use the override URL instead of the URL in certificates. The override URL is specified using the wolfSSL_CTX_SetOCSP_OverrideURL() function. WOLFSSL_OCSP_CHECKALL - Set all OCSP checks on WOLFSSL_OCSP_NO_NONCE - Set nonce option for creating OCSP requests.

	Name
int	**wolfSSL_CTX_DisableOCSP**(WOLFSSL_CTX * ctx)This function disables OCSP certificate revocation checking by affecting the ocspEnabled member of the WOLFSSL_CERT_MANAGER structure.
int	**wolfSSL_CTX_SetOCSP_OverrideURL**(WOLFSSL_CTX * ctx, const char * url)This function manually sets the URL for OCSP to use. By default, OCSP will use the URL found in the individual certificate unless the WOLFSSL_OCSP_URL_OVERRIDE option is set using the wolfSSL_CTX_EnableOCSP.
int	**wolfSSL_CTX_SetOCSP_Cb**(WOLFSSL_CTX * ctx, CbOCSPIO ioCb, CbOCSPRespFree respFreeCb, void * ioCbCtx)Sets the callback for the OCSP in the WOLFSSL_CTX structure.
int	**wolfSSL_CTX_EnableOCSPStapling**(WOLFSSL_CTX * ctx)This function enables OCSP stapling by calling wolfSSL_CertManagerEnableOCSPStapling().
void	**wolfSSL_KeepArrays**(WOLFSSL * ssl)Normally, at the end of the SSL handshake, wolfSSL frees temporary arrays. Calling this function before the handshake begins will prevent wolfSSL from freeing temporary arrays. Temporary arrays may be needed for things such as wolfSSL_get_keys() or PSK hints. When the user is done with temporary arrays, either wolfSSL_FreeArrays() may be called to free the resources immediately, or alternatively the resources will be freed when the associated SSL object is freed.
void	**wolfSSL_FreeArrays**(WOLFSSL * ssl)Normally, at the end of the SSL handshake, wolfSSL frees temporary arrays. If wolfSSL_KeepArrays() has been called before the handshake, wolfSSL will not free temporary arrays. This function explicitly frees temporary arrays and should be called when the user is done with temporary arrays and does not want to wait for the SSL object to be freed to free these resources.
int	**wolfSSL_UseSNI**(WOLFSSL * ssl, unsigned char type, const void * data, unsigned short size)This function enables the use of Server Name Indication in the SSL object passed in the 'ssl' parameter. It means that the SNI extension will be sent on ClientHello by wolfSSL client and wolfSSL server will respond ClientHello + SNI with either ServerHello + blank SNI or alert fatal in case of SNI mismatch.

	Name
int	**wolfSSL_CTX_UseSNI**(WOLFSSL_CTX * ctx, unsigned char type, const void * data, unsigned short size)This function enables the use of Server Name Indication for SSL objects created from the SSL context passed in the 'ctx' parameter. It means that the SNI extension will be sent on ClientHello by wolfSSL clients and wolfSSL servers will respond ClientHello + SNI with either ServerHello + blank SNI or alert fatal in case of SNI mismatch.
void	**wolfSSL_SNI_SetOptions**(WOLFSSL * ssl, unsigned char type, unsigned char options)This function is called on the server side to configure the behavior of the SSL session using Server Name Indication in the SSL object passed in the 'ssl' parameter. The options are explained below.
void	**wolfSSL_CTX_SNI_SetOptions**(WOLFSSL_CTX * ctx, unsigned char type, unsigned char options)This function is called on the server side to configure the behavior of the SSL sessions using Server Name Indication for SSL objects created from the SSL context passed in the 'ctx' parameter. The options are explained below.
int	**wolfSSL_SNI_GetFromBuffer**(const unsigned char * clientHello, unsigned int helloSz, unsigned char type, unsigned char * sni, unsigned int * inOutSz)This function is called on the server side to retrieve the Server Name Indication provided by the client from the Client Hello message sent by the client to start a session. It does not requires context or session setup to retrieve the SNI.
unsigned char	**wolfSSL_SNI_Status**(WOLFSSL * ssl, unsigned char type)This function gets the status of an SNI object.
unsigned short	**wolfSSL_SNI_GetRequest**(WOLFSSL * ssl, unsigned char type, void ** data)This function is called on the server side to retrieve the Server Name Indication provided by the client in a SSL session.
int	**wolfSSL_UseALPN**(WOLFSSL * ssl, char * protocol_name_list, unsigned int protocol_name_listSz, unsigned char options)Setup ALPN use for a wolfSSL session.
int	**wolfSSL_ALPN_GetProtocol**(WOLFSSL * ssl, char ** protocol_name, unsigned short * size)This function gets the protocol name set by the server.

	Name
int	**wolfSSL_ALPN_GetPeerProtocol**(WOLFSSL * ssl, char ** list, unsigned short * listSz)This function copies the alpn_client_list data from the SSL object to the buffer.
int	**wolfSSL_UseMaxFragment**(WOLFSSL * ssl, unsigned char mfl)This function is called on the client side to enable the use of Maximum Fragment Length in the SSL object passed in the 'ssl' parameter. It means that the Maximum Fragment Length extension will be sent on ClientHello by wolfSSL clients.
int	**wolfSSL_CTX_UseMaxFragment**(WOLFSSL_CTX * ctx, unsigned char mfl)This function is called on the client side to enable the use of Maximum Fragment Length for SSL objects created from the SSL context passed in the 'ctx' parameter. It means that the Maximum Fragment Length extension will be sent on ClientHello by wolfSSL clients.
int	**wolfSSL_UseTruncatedHMAC**(WOLFSSL * ssl)This function is called on the client side to enable the use of Truncated HMAC in the SSL object passed in the 'ssl' parameter. It means that the Truncated HMAC extension will be sent on ClientHello by wolfSSL clients.
int	**wolfSSL_CTX_UseTruncatedHMAC**(WOLFSSL_CTX * ctx)This function is called on the client side to enable the use of Truncated HMAC for SSL objects created from the SSL context passed in the 'ctx' parameter. It means that the Truncated HMAC extension will be sent on ClientHello by wolfSSL clients.
int	**wolfSSL_UseOCSPStapling**(WOLFSSL * ssl, unsigned char status_type, unsigned char options)Stapling eliminates the need to contact the CA. Stapling lowers the cost of certificate revocation check presented in OCSP.
int	**wolfSSL_CTX_UseOCSPStapling**(WOLFSSL_CTX * ctx, unsigned char status_type, unsigned char options)This function requests the certificate status during the handshake.
int	**wolfSSL_UseOCSPStaplingV2**(WOLFSSL * ssl, unsigned char status_type, unsigned char options)The function sets the status type and options for OCSP.
int	**wolfSSL_CTX_UseOCSPStaplingV2**(WOLFSSL_CTX * ctx, unsigned char status_type, unsigned char options)Creates and initializes the certificate status request for OCSP Stapling.

	Name
int	**wolfSSL_UseSupportedCurve**(WOLFSSL * ssl, word16 name)This function is called on the client side to enable the use of Supported Elliptic Curves Extension in the SSL object passed in the 'ssl' parameter. It means that the supported curves enabled will be sent on ClientHello by wolfSSL clients. This function can be called more than one time to enable multiple curves.
int	**wolfSSL_CTX_UseSupportedCurve**(WOLFSSL_CTX * ctx, word16 name)This function is called on the client side to enable the use of Supported Elliptic Curves Extension for SSL objects created from the SSL context passed in the 'ctx' parameter. It means that the supported curves enabled will be sent on ClientHello by wolfSSL clients. This function can be called more than one time to enable multiple curves.
int	**wolfSSL_UseSecureRenegotiation**(WOLFSSL * ssl)This function forces secure renegotiation for the supplied WOLFSSL structure. This is not recommended.
int	**wolfSSL_Rehandshake**(WOLFSSL * ssl)This function executes a secure renegotiation handshake; this is user forced as wolfSSL discourages this functionality.
int	**wolfSSL_UseSessionTicket**(WOLFSSL * ssl)Force provided WOLFSSL structure to use session ticket. The constant HAVE_SESSION_TICKET should be defined and the constant NO_WOLFSSL_CLIENT should not be defined to use this function.
int	**wolfSSL_CTX_UseSessionTicket**(WOLFSSL_CTX * ctx)This function sets wolfSSL context to use a session ticket.
int	**wolfSSL_get_SessionTicket**(WOLFSSL * ssl, unsigned char * buf, word32 * bufSz)This function copies the ticket member of the Session structure to the buffer. If buf is NULL and bufSz is non-NULL, bufSz will be set to the ticket length.
int	**wolfSSL_set_SessionTicket**(WOLFSSL * ssl, const unsigned char * buf, word32 bufSz)This function sets the ticket member of the WOLFSSL_SESSION structure within the WOLFSSL struct. The buffer passed into the function is copied to memory.

	Name
int	**wolfSSL_set_SessionTicket_cb**(WOLFSSL * ssl, CallbackSessionTicket cb, void * ctx)This function sets the session ticket callback. The type CallbackSessionTicket is a function pointer with the signature of: int (*CallbackSessionTicket)(WOLFSSL*, const unsigned char*, int, void*)
int	**wolfSSL_send_SessionTicket**(WOLFSSL * ssl)This function sends a session ticket to the client after a TLS v1.3 handhsake has been established.
int	**wolfSSL_CTX_set_TicketEncCb**(WOLFSSL_CTX * ctx, SessionTicketEncCb cb)This function sets the session ticket key encrypt callback function for a server to support session tickets as specified in RFC 5077.
int	**wolfSSL_CTX_set_TicketHint**(WOLFSSL_CTX * ctx, int hint)This function sets the session ticket hint relayed to the client. For server side use.
int	**wolfSSL_CTX_set_TicketEncCtx**(WOLFSSL_CTX * ctx, void * userCtx)This function sets the session ticket encrypt user context for the callback. For server side use.
void *	**wolfSSL_CTX_get_TicketEncCtx**(WOLFSSL_CTX * ctx)This function gets the session ticket encrypt user context for the callback. For server side use.
int	**wolfSSL_SetHsDoneCb**(WOLFSSL * ssl, HandShakeDoneCb cb, void * user_ctx)This function sets the handshake done callback. The hsDoneCb and hsDoneCtx members of the WOLFSSL structure are set in this function.
int	**wolfSSL_PrintSessionStats**(void)This function prints the statistics from the session.
int	**wolfSSL_get_session_stats**(unsigned int * active, unsigned int * total, unsigned int * peak, unsigned int * maxSessions)This function gets the statistics for the session.
int	**wolfSSL_MakeTlsMasterSecret**(unsigned char * ms, word32 msLen, const unsigned char * pms, word32 pmsLen, const unsigned char * cr, const unsigned char * sr, int tls1_2, int hash_type)This function copies the values of cr and sr then passes through to wc_PRF (pseudo random function) and returns that value.
int	**wolfSSL_DeriveTlsKeys**(unsigned char * key_data, word32 keyLen, const unsigned char * ms, word32 msLen, const unsigned char * sr, const unsigned char * cr, int tls1_2, int hash_type)An external facing wrapper to derive TLS Keys.

	Name
int	**wolfSSL_connect_ex**(WOLFSSL * ssl, HandShakeCallBack hsCb, TimeoutCallBack toCb, WOLFSSL_TIMEVAL timeout)wolfSSL_connect_ex() is an extension that allows a HandShake Callback to be set. This can be useful in embedded systems for debugging support when a debugger isn't available and sniffing is impractical. The HandShake Callback will be called whether or not a handshake error occurred. No dynamic memory is used since the maximum number of SSL packets is known. Packet names can be accessed through packetNames[]. The connect extension also allows a Timeout Callback to be set along with a timeout value. This is useful if the user doesn't want to wait for the TCP stack to timeout. This extension can be called with either, both, or neither callbacks.
int	**wolfSSL_accept_ex**(WOLFSSL * ssl, HandShakeCallBack hsCb, TimeoutCallBack toCb, WOLFSSL_TIMEVAL timeout)wolfSSL_accept_ex() is an extension that allows a HandShake Callback to be set. This can be useful in embedded systems for debugging support when a debugger isn't available and sniffing is impractical. The HandShake Callback will be called whether or not a handshake error occurred. No dynamic memory is used since the maximum number of SSL packets is known. Packet names can be accessed through packetNames[]. The connect extension also allows a Timeout Callback to be set along with a timeout value. This is useful if the user doesn't want to wait for the TCP stack to timeout. This extension can be called with either, both, or neither callbacks.
long	**wolfSSL_BIO_set_fp**(WOLFSSL_BIO * bio, XFILE fp, int c)This is used to set the internal file pointer for a BIO.
long	**wolfSSL_BIO_get_fp**(WOLFSSL_BIO * bio, XFILE * fp)This is used to get the internal file pointer for a BIO.
int	**wolfSSL_check_private_key**(const WOLFSSL * ssl)This function checks that the private key is a match with the certificate being used.
int	**wolfSSL_X509_get_ext_by_NID**(const WOLFSSL_X509 * x, int nid, int lastpos)This function looks for and returns the extension index matching the passed in NID value.

	Name
void *	**wolfSSL_X509_get_ext_d2i**(const WOLFSSL_X509 * x509, int nid, int * c, int * idx)This function looks for and returns the extension matching the passed in NID value.
int	**wolfSSL_X509_digest**(const WOLFSSL_X509 * x509, const WOLFSSL_EVP_MD * digest, unsigned char * buf, unsigned int * len)This function returns the hash of the DER certificate.
int	**wolfSSL_use_certificate**(WOLFSSL * ssl, WOLFSSL_X509 * x509)his is used to set the certificate for WOLFSSL structure to use during a handshake.
int	**wolfSSL_use_certificate_ASN1**(WOLFSSL * ssl, const unsigned char * der, int derSz)This is used to set the certificate for WOLFSSL structure to use during a handshake. A DER formatted buffer is expected.
int	**wolfSSL_use_PrivateKey**(WOLFSSL * ssl, WOLFSSL_EVP_PKEY * pkey)This is used to set the private key for the WOLFSSL structure.
int	**wolfSSL_use_PrivateKey_ASN1**(int pri, WOLFSSL * ssl, const unsigned char * der, long derSz)This is used to set the private key for the WOLFSSL structure. A DER formatted key buffer is expected.
int	**wolfSSL_use_RSAPrivateKey_ASN1**(WOLFSSL * ssl, unsigned char * der, long derSz)This is used to set the private key for the WOLFSSL structure. A DER formatted RSA key buffer is expected.
WOLFSSL_DH *	**wolfSSL_DSA_dup_DH**(const WOLFSSL_DSA * r)This function duplicates the parameters in dsa to a newly created WOLFSSL_DH structure.
int	**wolfSSL_SESSION_get_master_key**(const WOLFSSL_SESSION * ses, unsigned char * out, int outSz)This is used to get the master key after completing a handshake.
int	**wolfSSL_SESSION_get_master_key_length**(const WOLFSSL_SESSION * ses)This is used to get the master secret key length.
void	**wolfSSL_CTX_set_cert_store**(WOLFSSL_CTX * ctx, WOLFSSL_X509_STORE * str)This is a setter function for the WOLFSSL_X509_STORE structure in ctx.
WOLFSSL_X509 *	**wolfSSL_d2i_X509_bio**(WOLFSSL_BIO * bio, WOLFSSL_X509 ** x509)This function get the DER buffer from bio and converts it to a WOLFSSL_X509 structure.
WOLFSSL_X509_STORE *	**wolfSSL_CTX_get_cert_store**(WOLFSSL_CTX * ctx)This is a getter function for the WOLFSSL_X509_STORE structure in ctx.

	Name
size_t	**wolfSSL_BIO_ctrl_pending**(WOLFSSL_BIO * b)Gets the number of pending bytes to read. If BIO type is BIO_BIO then is the number to read from pair. If BIO contains an SSL object then is pending data from SSL object (wolfSSL_pending(ssl)). If is BIO_MEMORY type then returns the size of memory buffer.
size_t	**wolfSSL_get_server_random**(const WOLFSSL * ssl, unsigned char * out, size_t outlen)This is used to get the random data sent by the server during the handshake.
size_t	**wolfSSL_get_client_random**(const WOLFSSL * ssl, unsigned char * out, size_t outSz)This is used to get the random data sent by the client during the handshake.
wc_pem_password_cb *	**wolfSSL_CTX_get_default_passwd_cb**(WOLFSSL_CTX * ctx)This is a getter function for the password callback set in ctx.
void *	**wolfSSL_CTX_get_default_passwd_cb_userdata**(WOLFSSL_CTX * ctx)This is a getter function for the password callback user data set in ctx.
WOLFSSL_X509 *	**wolfSSL_PEM_read_bio_X509_AUX**(WOLFSSL_BIO * bp, WOLFSSL_X509 ** x, wc_pem_password_cb * cb, void * u)This function behaves the same as wolfSSL_PEM_read_bio_X509. AUX signifies containing extra information such as trusted/rejected use cases and friendly name for human readability.
long	**wolfSSL_CTX_set_tmp_dh**(WOLFSSL_CTX * ctx, WOLFSSL_DH * dh)Initializes the WOLFSSL_CTX structure's dh member with the Diffie-Hellman parameters.
WOLFSSL_DSA *	**wolfSSL_PEM_read_bio_DSAparams**(WOLFSSL_BIO * bp, WOLFSSL_DSA ** x, wc_pem_password_cb * cb, void * u)This function get the DSA parameters from a PEM buffer in bio.
unsigned long	**wolfSSL_ERR_peek_last_error**(void)This function returns the absolute value of the last error from WOLFSSL_ERROR encountered.
	WOLF_STACK_OF(WOLFSSL_X509) constThis function gets the peer's certificate chain.
long	**wolfSSL_CTX_clear_options**(WOLFSSL_CTX * ctx, long opt)This function resets option bits of WOLFSSL_CTX object.
int	**wolfSSL_set_jobject**(WOLFSSL * ssl, void * objPtr)This function sets the jObjectRef member of the WOLFSSL structure.
void *	**wolfSSL_get_jobject**(WOLFSSL * ssl)This function returns the jObjectRef member of the WOLFSSL structure.

	Name
int	**wolfSSL_set_msg_callback**(WOLFSSL * ssl, SSL_Msg_Cb cb)This function sets a callback in the ssl. The callback is to observe handshake messages. NULL value of cb resets the callback.
int	**wolfSSL_set_msg_callback_arg**(WOLFSSL * ssl, void * arg)This function sets associated callback context value in the ssl. The value is handed over to the callback argument.
char *	**wolfSSL_X509_get_next_altname**(WOLFSSL_X509 *)This function returns the next, if any, altname from the peer certificate.
WOLFSSL_ASN1_TIME *	**wolfSSL_X509_get_notBefore**(WOLFSSL_X509 *)The function checks to see if x509 is NULL and if it's not, it returns the notBefore member of the x509 struct.
int	**wolfSSL_connect**(WOLFSSL * ssl)This function is called on the client side and initiates an SSL/TLS handshake with a server. When this function is called, the underlying communication channel has already been set up. wolfSSL_connect() works with both blocking and non_blocking I/O. When the underlying I/O is non_blocking, wolfSSL_connect() will return when the underlying I/O could not satisfy the needs of wolfSSL_connect to continue the handshake. In this case, a call to wolfSSL_get_error() will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_connect() when the underlying I/O is ready and wolfSSL will pick up where it left off. When using a non_blocking socket, nothing needs to be done, but select() can be used to check for the required condition. If the underlying I/O is blocking, wolfSSL_connect() will only return once the handshake has been finished or an error occurred. wolfSSL takes a different approach to certificate verification than OpenSSL does. The default policy for the client is to verify the server, this means that if you don't load CAs to verify the server you'll get a connect error, unable to verify (_155). It you want to mimic OpenSSL behavior of having SSL_connect succeed even if verifying the server fails and reducing security you can do this by calling: SSL_CTX_set_verify(ctx, SSL_VERIFY_NONE, 0); before calling SSL_new(); Though it's not recommended.

	Name
int	**wolfSSL_send_hrr_cookie**(WOLFSSL * ssl, const unsigned char * secret, unsigned int secretSz)This function is called on the server side to indicate that a HelloRetryRequest message must contain a Cookie and, in case of using protocol DTLS v1.3, that the handshake will always include a cookie exchange. Please note that when using protocol DTLS v1.3, the cookie exchange is enabled by default. The Cookie holds a hash of the current transcript so that another server process can handle the ClientHello in reply. The secret is used when generating the integrity check on the Cookie data. This replaces or regenerates the HRR secret, then delegates to wolfSSL_enable_cookie(). Changing cookie mode after handshake processing starts is unsupported; rotating a secret while cookies are already enabled remains supported. DTLS 1.2_only callers should use wolfSSL_enable_cookie() and wolfSSL_DTLS_SetCookieSecret().
int	**wolfSSL_set_hrr_cookie_secret_secondary**(WOLFSSL * ssl, const unsigned char * secret, unsigned int secretSz)This function sets a secondary HelloRetryRequest cookie secret on a DTLS 1.3 server. It is used only when verifying a received cookie, and only if the primary secret (set with wolfSSL_send_hrr_cookie()) fails to verify it. This lets an application rotate the cookie secret on a stateless DTLS 1.3 server without rejecting clients whose cookie was issued under the previous secret: install the new secret as the primary and the previous secret here for an overlap window. The secondary secret is never used to issue cookies. This API is DTLS only; TLS 1.3 over a reliable transport does not verify cookies statelessly across the HelloRetryRequest exchange.

	Name
int	**wolfSSL_disable_hrr_cookie**(WOLFSSL * ssl)This function is called on the server side to indicate that a HelloRetryRequest message must NOT contain a Cookie and that, if using protocol DTLS v1.3, a cookie exchange will not be included in the handshake. Please note that not doing a cookie exchange when using protocol DTLS v1.3 can make the server susceptible to DoS/Amplification attacks. This delegates to wolfSSL_disable_cookie(): on a DTLS object it also disables the DTLS 1.2 HelloVerifyRequest exchange, and it fails with BAD_STATE_E once the handshake has committed to stateful processing.
int	**wolfSSL_CTX_no_ticket_TLSv13**(WOLFSSL_CTX * ctx)This function is called on the server to stop it from sending a resumption session ticket once the handshake is complete.
int	**wolfSSL_no_ticket_TLSv13**(WOLFSSL * ssl)This function is called on the server to stop it from sending a resumption session ticket once the handshake is complete.
int	**wolfSSL_CTX_no_dhe_psk**(WOLFSSL_CTX * ctx)This function is called on a TLS v1.3 wolfSSL context to disallow Diffie_Hellman (DH) style key exchanges when handshakes are using pre-shared keys for authentication.
int	**wolfSSL_no_dhe_psk**(WOLFSSL * ssl)This function is called on a TLS v1.3 client or server wolfSSL to disallow Diffie_Hellman (DH) style key exchanges when handshakes are using pre-shared keys for authentication.

	Name
int	**wolfSSL_CTX_require_psk**(WOLFSSL_CTX * ctx)This function is called on a TLS v1.3 / DTLS v1.3 context to require that an external Pre_Shared Key is negotiated for the handshake to succeed. When set, a handshake that completes without negotiating an external PSK is aborted with PSK_MISSING_ERROR instead of falling back to a certificate handshake, so the PSK acts as an additional security factor. The requirement keys off the external_PSK callback (it has no effect unless one is registered) and session_ticket resumption is exempt. To preserve forward secrecy a mandatory external PSK must also use an (EC)DHE key exchange; a pure psk_ke handshake is rejected with PSK_KEY_ERROR. This applies to TLS 1.3 and DTLS 1.3 only; in (D)TLS 1.2 the use of a PSK is determined by the negotiated cipher suite, so a mandatory PSK is instead configured by restricting the cipher suite list to (preferably (EC)DHE_)PSK suites.
int	**wolfSSL_require_psk**(WOLFSSL * ssl)This function is called on a TLS v1.3 / DTLS v1.3 wolfSSL object to require that an external Pre_Shared Key is negotiated for the handshake to succeed. See wolfSSL_CTX_require_psk() for the full behaviour.
int	**wolfSSL_update_keys**(WOLFSSL * ssl)This function is called on a TLS v1.3 client or server wolfSSL to force the rollover of keys. A KeyUpdate message is sent to the peer and new keys are calculated for encryption. The peer will send back a KeyUpdate message and the new decryption keys will then be calculated. This function can only be called after a handshake has been completed.
int	**wolfSSL_key_update_response**(WOLFSSL * ssl, int * required)This function is called on a TLS v1.3 client or server wolfSSL to determine whether a rollover of keys is in progress. When wolfSSL_update_keys() is called, a KeyUpdate message is sent and the encryption key is updated. The decryption key is updated when the response is received.
int	**wolfSSL_CTX_allow_post_handshake_auth**(WOLFSSL_CTX * ctx)This function is called on a TLS v1.3 client wolfSSL context to allow a client certificate to be sent post handshake upon request from server. This is useful when connecting to a web server that has some pages that require client authentication and others that don't.

	Name
int	**wolfSSL_allow_post_handshake_auth**(WOLFSSL * ssl)This function is called on a TLS v1.3 client wolfSSL to allow a client certificate to be sent post handshake upon request from server. A Post-Handshake Client Authentication extension is sent in the ClientHello. This is useful when connecting to a web server that has some pages that require client authentication and others that don't.
int	**wolfSSL_request_certificate**(WOLFSSL * ssl)This function requests a client certificate from the TLS v1.3 client. This is useful when a web server is serving some pages that require client authentication and others that don't. A maximum of 256 requests can be sent on a connection.
int	**wolfSSL_CTX_set1_sigalgs_list**(WOLFSSL_CTX * ctx, const char * list)This function sets the list of signature algorithms to allow on a wolfSSL context in order of preference. The list is a null_terminated text string and a colon_delimited list, where each element is of the form "+" (for example "RSA_PSS+SHA256:ECDSA+SHA384"). The Edwards-curve algorithms "ED25519" and "ED448" are written without a digest suffix, since the hash is implied by the algorithm. The previous list stored in the context is replaced.
int	**wolfSSL_set1_sigalgs_list**(WOLFSSL * ssl, const char * list)This function sets the list of signature algorithms to allow on a wolfSSL session in order of preference. The list format and the set of recognized public_key and digest tokens are identical to those documented for wolfSSL_CTX_set1_sigalgs_list(); refer to that function for the full token tables and TLS 1.3 caveats. The previous list stored in the session is replaced.
int	**wolfSSL_CTX_set1_groups_list**(WOLFSSL_CTX * ctx, const char * list)This function sets the list of key_exchange groups (named elliptic curves and KEMs) to allow on a wolfSSL context in order of preference. The list is a null_terminated, colon_delimited text string of group names, for example "P_384:P_256:X25519". Call this function to set the key_exchange parameters used with TLS v1.3 connections (the function is compiled in only when HAVE_ECC, WOLFSSL_TLS13, and HAVE_SUPPORTED_CURVES are defined).

	Name
int	**wolfSSL_set1_groups_list**(WOLFSSL * ssl, const char * list)This function sets the list of key_exchange groups (named elliptic curves and KEMs) to allow on a wolfSSL session in order of preference. The list format and the set of recognized group names are identical to those documented for wolfSSL_CTX_set1_groups_list(); refer to that function for the full token tables and required build options.
int	**wolfSSL_preferred_group**(WOLFSSL * ssl)This function returns the key exchange group the client prefers to use in the TLS v1.3 handshake. Call this function to after a handshake is complete to determine which group the server prefers so that this information can be used in future connections to pre-generate a key pair for key exchange.
int	**wolfSSL_CTX_set_groups**(WOLFSSL_CTX * ctx, int * groups, int count)This function sets the list of key_exchange groups (named elliptic curves and KEMs) to allow on a wolfSSL context in order of preference. The list is an array of named_group identifiers (see the table below) and count is the number of identifiers in the array. Use this function to set the key-exchange parameters used by TLS v1.3 connections; the order of the array becomes the local KeyShare preference order.
int	**wolfSSL_set_groups**(WOLFSSL * ssl, int * groups, int count)This function sets the list of key_exchange groups (named elliptic curves and KEMs) to allow on a wolfSSL session in order of preference. The array format and the set of recognized identifiers are identical to those documented for wolfSSL_CTX_set_groups(); refer to that function for the full identifier table and required build options.

	Name
int	**wolfSSL_connect_TLSv13**(WOLFSSL * ssl)This function is called on the client side and initiates a TLS v1.3 handshake with a server. When this function is called, the underlying communication channel has already been set up. wolfSSL_connect() works with both blocking and non_blocking I/O. When the underlying I/O is non_blocking, wolfSSL_connect() will return when the underlying I/O could not satisfy the needs of wolfSSL_connect to continue the handshake. In this case, a call to wolfSSL_get_error() will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_connect() when the underlying I/O is ready and wolfSSL will pick up where it left off. When using a non_blocking socket, nothing needs to be done, but select() can be used to check for the required condition. If the underlying I/O is blocking, wolfSSL_connect() will only return once the handshake has been finished or an error occurred. wolfSSL takes a different approach to certificate verification than OpenSSL does. The default policy for the client is to verify the server, this means that if you don't load CAs to verify the server you'll get a connect error, unable to verify (_155). It you want to mimic OpenSSL behavior of having SSL_connect succeed even if verifying the server fails and reducing security you can do this by calling: SSL_CTX_set_verify(ctx, SSL_VERIFY_NONE, 0); before calling SSL_new(); Though it's not recommended.

	Name
	wolfSSL_accept_TLSv13(WOLFSSL * ssl)This function is called on the server side and waits for a SSL/TLS client to initiate the SSL/TLS handshake. When this function is called, the underlying communication channel has already been set up. wolfSSL_accept() works with both blocking and non_blocking I/O. When the underlying I/O is non_blocking, wolfSSL_accept() will return when the underlying I/O could not satisfy the needs of wolfSSL_accept to continue the handshake. In this case, a call to wolfSSL_get_error() will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_accept when data is available to read and wolfSSL will pick up where it left off. When using a non_blocking socket, nothing needs to be done, but select() can be used to check for the required condition. If the underlying I/O is blocking, wolfSSL_accept() will only return once the handshake has been finished or an error occurred. Call this function when expecting a TLS v1.3 connection though older version ClientHello messages are supported.
int	**wolfSSL_CTX_set_max_early_data**(WOLFSSL_CTX * ctx, unsigned int sz)This function sets the maximum amount of early data that a TLS v1.3 client or server is willing to exchange using the wolfSSL context. Call this function to limit the amount of early data to process to mitigate replay attacks. Early data is protected by keys derived from those of the connection that the session ticket was sent and therefore will be the same every time a session ticket is used in resumption. The value is included in the session ticket for resumption. A server value of zero indicates no early data is to be sent by client using session tickets. A client value of zero indicates that the client will not send any early data. The default value is zero: per RFC 8446 Appendix E.5, TLS implementations “MUST NOT enable 0_RTT (either sending or accepting) unless specifically requested by the application.” Servers must call this function (or the per_SSL equivalent) with a non-zero value to opt in. It is recommended that the number of early data bytes be kept as low as practically possible in the application.

	Name
int	**wolfSSL_set_max_early_data**(WOLFSSL * ssl, unsigned int sz)This function sets the maximum amount of early data that a TLS v1.3 client or server is willing to exchange. Call this function to limit the amount of early data to process to mitigate replay attacks. Early data is protected by keys derived from those of the connection that the session ticket was sent and therefore will be the same every time a session ticket is used in resumption. The value is included in the session ticket for resumption. A server value of zero indicates no early data is to be sent by client using session tickets. A client value of zero indicates that the client will not send any early data. It is recommended that the number of early data bytes be kept as low as practically possible in the application.
int	**wolfSSL_write_early_data**(WOLFSSL * ssl, const void * data, int sz, int * outSz)This function writes early data to the server on resumption. Call this function before wolfSSL_connect() or wolfSSL_connect_TLSv13(). This function is only used with clients.
int	**wolfSSL_read_early_data**(WOLFSSL * ssl, void * data, int sz, int * outSz)This function reads any early data from a client on resumption. Call this function instead of wolfSSL_accept() or wolfSSL_accept_TLSv13() to accept a client and read any early data in the handshake. The function should be invoked until wolfSSL_is_init_finished() returns true. Early data may be sent by the client in multiple messages. If there is no early data then the handshake will be processed as normal. This function is only used with servers.
int	**wolfSSL_CTX_no_early_data_fresh_start_check**(WOLFSSL_CTX * ctx)This function is called on the server to disable the RFC 8446 Section 8.2 fresh start protection. By default a freshly created context rejects early data, but not resumption, for session tickets minted before the context was created, since the anti_replay state for those tickets may not have survived a server restart. Only call this function when the anti_replay state (session cache or external cache) reliably survives server restarts.

COPYRIGHT ©2024 wolfSSL Inc.

	Name
int	**wolfSSL_inject**(WOLFSSL * ssl, const void * data, int sz)This function is called to inject data into the WOLFSSL object. This is useful when data needs to be read from a single place and demultiplexed into multiple connections. The caller should then call wolfSSL_read() to extract the plaintext data from the WOLFSSL object.
void	**wolfSSL_CTX_set_psk_client_tls13_callback**(WOLFSSL_CTX * ctx, wc_psk_client_tls13_callback cb)This function sets the Pre_Shared Key (PSK) client side callback for TLS v1.3 connections. The callback is used to find a PSK identity and return its key and the name of the cipher to use for the handshake. The function sets the client_psk_tls13_cb member of the WOLFSSL_CTX structure.
void	**wolfSSL_set_psk_client_tls13_callback**(WOLFSSL * ssl, wc_psk_client_tls13_callback cb)This function sets the Pre_Shared Key (PSK) client side callback for TLS v1.3 connections. The callback is used to find a PSK identity and return its key and the name of the cipher to use for the handshake. The function sets the client_psk_tls13_cb member of the options field in WOLFSSL structure.
void	**wolfSSL_CTX_set_psk_server_tls13_callback**(WOLFSSL_CTX * ctx, wc_psk_server_tls13_callback cb)This function sets the Pre_Shared Key (PSK) server side callback for TLS v1.3 connections. The callback is used to find a PSK identity and return its key and the name of the cipher to use for the handshake. The function sets the server_psk_tls13_cb member of the WOLFSSL_CTX structure.
void	**wolfSSL_set_psk_server_tls13_callback**(WOLFSSL * ssl, wc_psk_server_tls13_callback cb)This function sets the Pre_Shared Key (PSK) server side callback for TLS v1.3 connections. The callback is used to find a PSK identity and return its key and the name of the cipher to use for the handshake. The function sets the server_psk_tls13_cb member of the options field in WOLFSSL structure.
int	**wolfSSL_CTX_set_cert_with_extern_psk**(WOLFSSL_CTX * ctx, int state)Enable or disable TLS 1.3 certificate authentication with external PSK (RFC 9973) on a context.
int	**wolfSSL_set_cert_with_extern_psk**(WOLFSSL * ssl, int state)Enable or disable TLS 1.3 certificate authentication with external PSK (RFC 9973) on a connection.

	Name
int	**wolfSSL_UseKeyShare**(WOLFSSL * ssl, word16 group)This function creates a key share entry from the group including generating a key pair. The KeyShare extension contains all the generated public keys for key exchange. If this function is called, then only the groups specified will be included. Call this function when a preferred group has been previously established for the server.
int	**wolfSSL_NoKeyShares**(WOLFSSL * ssl)This function is called to ensure no key shares are sent in the ClientHello. This will force the server to respond with a HelloRetryRequest if a key exchange is required in the handshake. Call this function when the expected key exchange group is not known and to avoid the generation of keys unnecessarily. Note that an extra round-trip will be required to complete the handshake when a key exchange is required.
WOLFSSL_METHOD *	**wolfTLSv1_3_server_method_ex**(void * heap)This function is used to indicate that the application is a server and will only support the TLS 1.3 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().
WOLFSSL_METHOD *	**wolfTLSv1_3_client_method_ex**(void * heap)This function is used to indicate that the application is a client and will only support the TLS 1.3 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().
WOLFSSL_METHOD *	**wolfTLSv1_3_server_method**(void)This function is used to indicate that the application is a server and will only support the TLS 1.3 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().
WOLFSSL_METHOD *	**wolfTLSv1_3_client_method**(void)This function is used to indicate that the application is a client and will only support the TLS 1.3 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

	Name
WOLFSSL_METHOD *	**wolfTLSv1_3_method_ex**(void * heap)This function returns a WOLFSSL_METHOD similar to wolfTLSv1_3_client_method except that it is not determined which side yet (server/client).
WOLFSSL_METHOD *	**wolfTLSv1_3_method**(void)This function returns a WOLFSSL_METHOD similar to wolfTLSv1_3_client_method except that it is not determined which side yet (server/client).
int	**wolfSSL_CTX_set_ephemeral_key**(WOLFSSL_CTX * ctx, int keyAlgo, const char * key, unsigned int keySz, int format)This function sets a fixed / static ephemeral key for testing only.
int	**wolfSSL_set_ephemeral_key**(WOLFSSL * ssl, int keyAlgo, const char * key, unsigned int keySz, int format)This function sets a fixed / static ephemeral key for testing only.
int	**wolfSSL_CTX_get_ephemeral_key**(WOLFSSL_CTX * ctx, int keyAlgo, const unsigned char ** key, unsigned int * keySz)This function returns pointer to loaded key as ASN.1/DER.
int	**wolfSSL_get_ephemeral_key**(WOLFSSL * ssl, int keyAlgo, const unsigned char ** key, unsigned int * keySz)This function returns pointer to loaded key as ASN.1/DER.
int	**wolfSSL_RSA_sign_generic_padding**(int hashAlg, const unsigned char * hash, unsigned int hLen, unsigned char * sigRet, unsigned int * sigLen, WOLFSSL_RSA * rsa, int flag, int padding)Sign a message with the chosen message digest, padding, and RSA key.
int	**wolfSSL_dtls13_has_pending_msg**(WOLFSSL * ssl)checks if DTLSv1.3 stack has some messages sent but not yet acknowledged by the other peer
unsigned int	**wolfSSL_SESSION_get_max_early_data**(const WOLFSSL_SESSION * s)Get the maximum size of Early Data from a session.
int	**wolfSSL_CRYPTO_get_ex_new_index**(int class_index, long argl, void * argp, WOLFSSL_CRYPTO_EX_new * new_func, WOLFSSL_CRYPTO_EX_dup * dup_func, WOLFSSL_CRYPTO_EX_free * free_func)Get a new index for external data. This entry applies also for the following API:

	Name
int	**wolfSSL_CTX_set_client_cert_type**(WOLFSSL_CTX * ctx, const char * buf, int len)In case this function is called in a client side, set certificate types that can be sent to its peer. In case called in a server side, set certificate types that can be acceptable from its peer. Put cert types in the buffer with prioritised order. To reset the settings to default, pass NULL for the buffer or pass zero for len. By default, certificate type is only X509. In case both side intend to send or accept “Raw public key” cert, WOLFSSL_CERT_TYPE_RPK should be included in the buffer to set.
int	**wolfSSL_CTX_set_server_cert_type**(WOLFSSL_CTX * ctx, const char * buf, int len)In case this function is called in a server side, set certificate types that can be sent to its peer. In case called in a client side, set certificate types that can be acceptable from its peer. Put cert types in the buffer with prioritised order. To reset the settings to default, pass NULL for the buffer or pass zero for len. By default, certificate type is only X509. In case both side intend to send or accept “Raw public key” cert, WOLFSSL_CERT_TYPE_RPK should be included in the buffer to set.
int	**wolfSSL_set_client_cert_type**(WOLFSSL * ssl, const char * buf, int len)In case this function is called in a client side, set certificate types that can be sent to its peer. In case called in a server side, set certificate types that can be acceptable from its peer. Put cert types in the buffer with prioritised order. To reset the settings to default, pass NULL for the buffer or pass zero for len. By default, certificate type is only X509. In case both side intend to send or accept “Raw public key” cert, WOLFSSL_CERT_TYPE_RPK should be included in the buffer to set.
int	**wolfSSL_set_server_cert_type**(WOLFSSL * ssl, const char * buf, int len)In case this function is called in a server side, set certificate types that can be sent to its peer. In case called in a client side, set certificate types that can be acceptable from its peer. Put cert types in the buffer with prioritised order. To reset the settings to default, pass NULL for the buffer or pass zero for len. By default, certificate type is only X509. In case both side intend to send or accept “Raw public key” cert, WOLFSSL_CERT_TYPE_RPK should be included in the buffer to set.

	Name
int	**wolfSSL_CTX_clear_group_messages**(WOLFSSL_CTX * ctx)Disables handshake message grouping for the given WOLFSSL_CTX context.
int	**wolfSSL_clear_group_messages**(WOLFSSL * ssl)Disables handshake message grouping for the given WOLFSSL object.
int	**wolfSSL_get_negotiated_client_cert_type**(WOLFSSL * ssl, int * tp)This function returns the result of the client certificate type negotiation done in ClientHello and ServerHello. WOLFSSL_SUCCESS is returned as a return value if no negotiation occurs and WOLFSSL_CERT_TYPE_UNKNOWN is returned as the certificate type.
int	**wolfSSL_get_negotiated_server_cert_type**(WOLFSSL * ssl, int * tp)This function returns the result of the server certificate type negotiation done in ClientHello and ServerHello. WOLFSSL_SUCCESS is returned as a return value if no negotiation occurs and WOLFSSL_CERT_TYPE_UNKNOWN is returned as the certificate type.
int	**wolfSSL_CTX_set_expected_rpk**(WOLFSSL_CTX * ctx, const unsigned char * spki, unsigned int spkiSz)Pin a DER_encoded SubjectPublicKeyInfo that the peer is expected to present as a Raw Public Key (RFC 7250), establishing out_of_band trust on the WOLFSSL_CTX object. An unauthenticated RPK peer is otherwise rejected: when the peer is being authenticated (any verify mode other than WOLFSSL_VERIFY_NONE) the handshake fails closed unless the presented key matches a pin (or a verify callback accepts it). May be called more than once to pin several keys, up to WOLFSSL_MAX_RPK_PINS. The key is stored as its SHA_256 digest, so this API requires SHA_256. Pins are append_only _ there is no per_entry remove, but wolfSSL_CTX_clear_expected_rpk() empties the table so it can be repopulated.

	Name
int	**wolfSSL_set_expected_rpk**(WOLFSSL * ssl, const unsigned char * spki, unsigned int spkiSz)Pin a DER_encoded SubjectPublicKeyInfo that the peer is expected to present as a Raw Public Key (RFC 7250), establishing out_of_band trust on the WOLFSSL object. An unauthenticated RPK peer is otherwise rejected: when the peer is being authenticated (any verify mode other than WOLFSSL_VERIFY_NONE) the handshake fails closed unless the presented key matches a pin (or a verify callback accepts it). May be called more than once to pin several keys, up to WOLFSSL_MAX_RPK_PINS. The key is stored as its SHA_256 digest, so this API requires SHA_256. Pins are append_only _ there is no per_entry remove, but wolfSSL_clear_expected_rpk() empties the table so it can be repopulated.
int	**wolfSSL_CTX_clear_expected_rpk**(WOLFSSL_CTX * ctx)Remove all pinned expected peer Raw Public Keys (RFC 7250) from the WOLFSSL_CTX object, emptying the table so it can be repopulated _ for example across a peer key rotation, since the pinning APIs are otherwise append_only. Like other WOLFSSL_CTX setters this is not locked, so reconfigure the pins on a shared CTX while no handshakes are in flight (a WOLFSSL created with wolfSSL_new() copies the pin table by value at creation time).
int	**wolfSSL_clear_expected_rpk**(WOLFSSL * ssl)Remove all pinned expected peer Raw Public Keys (RFC 7250) from the WOLFSSL object, emptying the table so it can be repopulated - for example across a peer key rotation, since the pinning APIs are otherwise append-only.
int	**wolfSSL_dtls_cid_use**(WOLFSSL * ssl)Enable use of ConnectionID extensions for the SSL object. See RFC 9146 and RFC 9147.
int	**wolfSSL_dtls_cid_is_enabled**(WOLFSSL * ssl)If invoked after the handshake is complete it checks if ConnectionID was successfully negotiated for the SSL object. See RFC 9146 and RFC 9147.

	Name
int	**wolfSSL_dtls_cid_set**(WOLFSSL * ssl, unsigned char * cid, unsigned int size)Set the ConnectionID used by the other peer to send records in this connection. See RFC 9146 and RFC 9147. The ConnectionID must be at maximum DTLS_CID_MAX_SIZE, that is an tunable compile time define, and it can't never be bigger than 255 bytes.
int	**wolfSSL_dtls_cid_get_rx_size**(WOLFSSL * ssl, unsigned int * size)Get the size of the ConnectionID used by the other peer to send records in this connection. See RFC 9146 and RFC 9147. The size is stored in the parameter size.
int	**wolfSSL_dtls_cid_get_rx**(WOLFSSL * ssl, unsigned char * buffer, unsigned int bufferSz)Copy the ConnectionID used by the other peer to send records in this connection into the buffer pointed by the parameter buffer. See RFC 9146 and RFC.
int	**wolfSSL_dtls_cid_get0_rx**(WOLFSSL * ssl, unsigned char ** cid)Get the ConnectionID used by the other peer. See RFC 9146 and RFC 9147.
int	**wolfSSL_dtls_cid_get_tx_size**(WOLFSSL * ssl, unsigned int * size)Get the size of the ConnectionID used to send records in this connection. See RFC 9146 and RFC 9147. The size is stored in the parameter size. The send ConnectionID is chosen by the peer. On DTLS 1.3 it may be up to 255 bytes and is not bounded by wolfSSL_dtls_cid_max_size(); on DTLS 1.2 it never exceeds wolfSSL_dtls_cid_max_size().
int	**wolfSSL_dtls_cid_get_tx**(WOLFSSL * ssl, unsigned char * buffer, unsigned int bufferSz)Copy the ConnectionID used when sending records in this connection into the buffer pointer by the parameter buffer. See RFC 9146 and RFC 9147. The available size need to be provided in bufferSz. The send ConnectionID is chosen by the peer. On DTLS 1.3 it may be up to 255 bytes and is not bounded by wolfSSL_dtls_cid_max_size(), so size the buffer from wolfSSL_dtls_cid_get_tx_size() (or use 255 bytes) rather than from wolfSSL_dtls_cid_max_size(), otherwise this returns LENGTH_ERROR.

	Name
int	**wolfSSL_dtls_cid_get0_tx**(WOLFSSL * ssl, unsigned char ** cid)Get the ConnectionID used when sending records in this connection. See RFC 9146 and RFC 9147. The send ConnectionID is chosen by the peer. On DTLS 1.3 it may be up to 255 bytes and is not bounded by wolfSSL_dtls_cid_max_size(); use wolfSSL_dtls_cid_get_tx_size() to learn its length.
int	**wolfSSL_dtls_cid_max_size**(void)Get the maximum size of the ConnectionID that this build can receive, that is the largest size accepted by wolfSSL_dtls_cid_set(). This is the compile time define DTLS_CID_MAX_SIZE and it can never be bigger than 255 bytes. It does not bound the ConnectionID used to send records, which is chosen by the peer and, on DTLS 1.3, may be up to 255 bytes.
const unsigned char *	**wolfSSL_dtls_cid_parse**(const unsigned char * msg, unsigned int msgSz, unsigned int cidSz)Extract the ConnectionID from a record datagram/message. See RFC 9146 and RFC 9147.
void	**wolfSSL_CTX_set_client_CA_list**(WOLFSSL_CTX * ctx, WOLF_STACK_OF(WOLFSSL_X509_NAME) * names)On the server, this sets a list of CA names to be sent to clients in certificate requests as a hint for which CA's are supported by the server.
WOLFSSL_STACK *	**wolfSSL_CTX_get_client_CA_list**(const WOLFSSL_CTX * ctx)This retrieves the list previously set via wolfSSL_CTX_set_client_CA_list, or NULL if no list has been set.
void	**wolfSSL_set_client_CA_list**(WOLFSSL * ssl, WOLF_STACK_OF(WOLFSSL_X509_NAME) * names)Same as wolfSSL_CTX_set_client_CA_list, but specific to a session. If a CA list is set on both the context and the session, the list on the session is used.
WOLFSSL_STACK *	**wolfSSL_get_client_CA_list**(const WOLFSSL * ssl)On the server, this retrieves the list previously set via wolfSSL_set_client_CA_list. If none was set, returns the list previously set via wolfSSL_CTX_set_client_CA_list. If no list at all was set, returns NULL.
void	**wolfSSL_CTX_set0_CA_list**(WOLFSSL_CTX * ctx, WOLF_STACK_OF(WOLFSSL_X509_NAME) * names)This function sets a list of CA names to be sent to the peer as a hint for which CA's are supported for its authentication.

	Name
WOLFSSL_STACK *	**wolfSSL_CTX_get0_CA_list**(const WOLFSSL_CTX * ctx)This retrieves the list previously set via wolfSSL_CTX_set0_CA_list, or NULL if no list has been set.
void	**wolfSSL_set0_CA_list**(WOLFSSL * ssl, WOLF_STACK_OF(WOLFSSL_X509_NAME) * names)Same as wolfSSL_CTX_set0_CA_list, but specific to a session. If a CA list is set on both the context and the session, the list on the session is used.
WOLFSSL_STACK *	**wolfSSL_get0_CA_list**(const WOLFSSL * ssl)This retrieves the list previously set via wolfSSL_set0_CA_list. If none was set, returns the list previously set via wolfSSL_CTX_set0_CA_list. If no list at all was set, returns NULL.
WOLFSSL_STACK *	**wolfSSL_get0_peer_CA_list**(const WOLFSSL * ssl)This returns the CA list received from the peer.
int	**wolfSSL_get_client_suites_sigalgs**(const WOLFSSL * ssl, const byte ** suites, word16 * suiteSz, const byte ** hashSigAlgo, word16 * hashSigAlgoSz)This function returns the raw list of ciphersuites and signature algorithms offered by the client. The lists are only stored and returned inside a callback setup with wolfSSL_CTX_set_cert_cb(). This is useful to be able to dynamically load certificates and keys based on the available ciphersuites and signature algorithms.
WOLFSSL_CIPHERSUITE_INFO	**wolfSSL_get_ciphersuite_info**(byte first, byte second)This returns information about the ciphersuite directly from the raw ciphersuite bytes.
int	**wolfSSL_get_sigalg_info**(byte first, byte second, int * hashAlgo, int * sigAlgo)This returns information about the hash and signature algorithm directly from the raw ciphersuite bytes.
void	**wolfSSL_CTX_set_default_passwd_cb**(WOLFSSL_CTX * ctx, wc_pem_password_cb * cb)This function will set the password callback in the provided CTX. This callback is used when loading an encrypted cert or key which requires a password.
void	**wolfSSL_CTX_set_default_passwd_cb_userdata**(WOLFSSL_CTX * ctx, void * userdata)This function will set the userdata argument to the passwd_userdata member of the WOLFSSL_CTX structure. This member is passed into the CTX's password callback when called.

	Name
int	**wolfSSL_get_scr_check_enabled**(const WOLFSSL * ssl)Gets the state of the secure renegotiation (SCR) check requirement.
int	**wolfSSL_set_scr_check_enabled**(WOLFSSL * ssl, byte enabled)Sets the state of the secure renegotiation (SCR) check requirement.
int	**wolfSSL_CTX_DisableExtendedMasterSecret**(WOLFSSL_CTX * ctx)Disables the TLS Extended Master Secret extension (RFC 7627) on the context: a client stops advertising it and a server ignores the peer's request, so a standard master secret is negotiated. A server also declines resumption of sessions or tickets that used EMS and does a full handshake instead. TLS 1.2 and earlier only. Requires HAVE_EXTENDED_MASTER.
int	**wolfSSL_DisableExtendedMasterSecret**(WOLFSSL * ssl)Disables the TLS Extended Master Secret extension (RFC 7627) on the SSL object: a client stops advertising it and a server ignores the peer's request, so a standard master secret is negotiated. A server also declines resumption of sessions or tickets that used EMS and does a full handshake instead. Call before the handshake starts, or after wolfSSL_clear. TLS 1.2 and earlier only. Requires HAVE_EXTENDED_MASTER.
int	**wolfSSL_CTX_EnableExtendedMasterSecret**(WOLFSSL_CTX * ctx)Re_enables the TLS Extended Master Secret extension (RFC 7627) on the context (the default): EMS is used when the peer supports it but is not mandatory. Undoes a previous disable or require. Requires HAVE_EXTENDED_MASTER.
int	**wolfSSL_EnableExtendedMasterSecret**(WOLFSSL * ssl)Re_enables the TLS Extended Master Secret extension (RFC 7627) on the SSL object (the default): EMS is used when the peer supports it but is not mandatory. Undoes a previous disable or require. Call before the handshake starts, or after wolfSSL_clear. Requires HAVE_EXTENDED_MASTER.
int	**wolfSSL_CTX_RequireExtendedMasterSecret**(WOLFSSL_CTX * ctx)Makes the TLS Extended Master Secret extension (RFC 7627) mandatory on the context: if it is not negotiated, the connection is aborted with EXT_MASTER_SECRET_NEEDED_E. A client advertises the extension even after a previous disable. TLS 1.2 and earlier only. Requires HAVE_EXTENDED_MASTER.

	Name
int	**wolfSSL_RequireExtendedMasterSecret**(WOLFSSL * ssl)Makes the TLS Extended Master Secret extension (RFC 7627) mandatory on the SSL object: if it is not negotiated, including on resumption, the connection is aborted with EXT_MASTER_SECRET_NEEDED_E. A client advertises the extension even after a previous disable. Call before the handshake starts, or after wolfSSL_clear. TLS 1.2 and earlier only. Requires HAVE_EXTENDED_MASTER.
int	**wolfSSL_UseCertificateAuthority**(WOLFSSL * ssl, const unsigned char * dn, unsigned int dnSz)Adds a CA distinguished name to the list of certificate authorities announced via the TLS 1.3 certificate_authorities extension (RFC 8446 section 4.2.4) on the given SSL session. The DN must be the inner content of a DER_encoded X.509 Name (the bytes after the SEQUENCE tag and length), as returned by wc_GetDecodedCertSubjectRaw(). The library copies the bytes and prepends the SEQUENCE header on the wire automatically.
int	**wolfSSL_CTX_UseCertificateAuthority**(WOLFSSL_CTX * ctx, const unsigned char * dn, unsigned int dnSz)Adds a CA distinguished name to the list of certificate authorities announced via the TLS 1.3 certificate_authorities extension (RFC 8446 section 4.2.4) on all SSL sessions created from this context. The DN format and requirements are identical to wolfSSL_UseCertificateAuthority().
void	**wolfSSL_ClearCertificateAuthorities**(WOLFSSL * ssl)Frees and removes all CA distinguished names previously added to the SSL session via wolfSSL_UseCertificateAuthority(). After this call the session_level native CA list is empty; the CTX_level list (if any) is not affected.
void	**wolfSSL_CTX_ClearCertificateAuthorities**(WOLFSSL_CTX * ctx)Frees and removes all CA distinguished names previously added to the context via wolfSSL_CTX_UseCertificateAuthority(). After this call the CTX-level native CA list is empty.
int	**wolfSSL_GetPeerCertificateAuthorityCount**(const WOLFSSL * ssl)Returns the number of CA distinguished names received from the peer in the TLS 1.3 certificate_authorities extension. This is typically called inside a cert_cb (WOLFSSL_CERT_SETUP_CB) on the server side to inspect which CAs the client trusts.

	Name
int	**wolfSSL_GetPeerCertificateAuthority**(const WOLFSSL * ssl, int idx, unsigned char * outDn, unsigned int outDnSz)Copies the idx_th CA distinguished name received from the peer in the TLS 1.3 certificate_authorities extension into the caller's buffer. The DN is the inner content of the DER_encoded Name (without the SEQUENCE header), matching the format accepted by wolfSSL_UseCertificateAuthority().

C.58.3 Types Documentation

C.58.3.1 typedef WOLFSSL_TLSEXT_DEBUG_CB

```
typedef void(* WOLFSSL_TLSEXT_DEBUG_CB) (WOLFSSL *ssl, int client_server, int
 ↪  type, const byte *data, int len, void *arg);
```

Callback type for the TLS extension debug callback.

Parameters:

- **ssl** The WOLFSSL object receiving the extension.
- **client_server** 1 if the WOLFSSL object is a client, 0 if a server.
- **type** The extension type, e.g. TLSX_SERVER_NAME.
- **data** The raw extension content (data after the 2-byte length).
- **len** Length of the extension content in bytes.
- **arg** The argument set with wolfSSL_set_tlsext_debug_arg().

Invoked once for every TLS extension received during the handshake, in wire order, before the extension is processed.

Note that, unlike OpenSSL 3.x, the callback also reports unknown (unregistered) extension types.

C.58.4 Functions Documentation

C.58.4.1 function wolfDTLSv1_2_client_method_ex

```
WOLFSSL_METHOD * wolfDTLSv1_2_client_method_ex(
    void * heap
)
```

This function initializes the DTLS v1.2 client method.

Parameters:

- **heap** pointer to a heap hint for memory allocation.

See:

- wolfSSL_Init
- wolfSSL_CTX_new

Return: pointer This function returns a pointer to a new WOLFSSL_METHOD structure.

Example

```
wolfSSL_Init();
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(wolfDTLSv1_2_client_method());
…
```

```
WOLFSSL* ssl = wolfSSL_new(ctx);
…
```

C.58.4.2 function wolfSSLv23_method

```
WOLFSSL_METHOD * wolfSSLv23_method(
    void
)
```

This function returns a WOLFSSL_METHOD similar to wolfSSLv23_client_method except that it is not determined which side yet (server/client).

Parameters:

- **none** No parameters.

See:

- wolfSSL_new
- wolfSSL_free

Return:

- WOLFSSL_METHOD* On successful creations returns a WOLFSSL_METHOD pointer
- NULL Null if memory allocation error or failure to create method

Example

```
WOLFSSL* ctx;
ctx  = wolfSSL_CTX_new(wolfSSLv23_method());
// check ret value
```

C.58.4.3 function wolfSSLv3_server_method

```
WOLFSSL_METHOD * wolfSSLv3_server_method(
    void
)
```

The wolfSSLv3_server_method() function is used to indicate that the application is a server and will only support the SSL 3.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Parameters:

- **none** No parameters.

See:

- wolfTLSv1_server_method
- wolfTLSv1_1_server_method
- wolfTLSv1_2_server_method
- wolfTLSv1_3_server_method
- wolfDTLSv1_server_method
- wolfSSLv23_server_method
- wolfSSL_CTX_new

Return:

- – If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfSSLv3_server_method();
if (method == NULL) {
    unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.58.4.4 function wolfSSLv3_client_method

```
WOLFSSL_METHOD * wolfSSLv3_client_method(
    void
)
```

The wolfSSLv3_client_method() function is used to indicate that the application is a client and will only support the SSL 3.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Parameters:

- **none** No parameters.

See:

- wolfTLSv1_client_method
- wolfTLSv1_1_client_method
- wolfTLSv1_2_client_method
- wolfTLSv1_3_client_method
- wolfDTLSv1_client_method
- wolfSSLv23_client_method
- wolfSSL_CTX_new

Return:

- – If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfSSLv3_client_method();
if (method == NULL) {
    unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.58.4.5 function wolfTLSv1_server_method

```
WOLFSSL_METHOD * wolfTLSv1_server_method(
    void
)
```

The wolfTLSv1_server_method() function is used to indicate that the application is a server and will only support the TLS 1.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Parameters:

- **none** No parameters.

See:

- wolfSSLv3_server_method
- wolfTLSv1_1_server_method
- wolfTLSv1_2_server_method
- wolfTLSv1_3_server_method
- wolfDTLSv1_server_method
- wolfSSLv23_server_method
- wolfSSL_CTX_new

Return:

- – If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_server_method();
if (method == NULL) {
    unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.58.4.6 function wolfTLSv1_client_method

```
WOLFSSL_METHOD * wolfTLSv1_client_method(
    void
)
```

The wolfTLSv1_client_method() function is used to indicate that the application is a client and will only support the TLS 1.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Parameters:

- **none** No parameters.

See:

- wolfSSLv3_client_method
- wolfTLSv1_1_client_method
- wolfTLSv1_2_client_method
- wolfTLSv1_3_client_method
- wolfDTLSv1_client_method
- wolfSSLv23_client_method
- wolfSSL_CTX_new

Return:

- – If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_client_method();
if (method == NULL) {
    unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.58.4.7 function wolfTLSv1_1_server_method

```
WOLFSSL_METHOD * wolfTLSv1_1_server_method(
    void
)
```

The wolfTLSv1_1_server_method() function is used to indicate that the application is a server and will only support the TLS 1.1 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Parameters:

- **none** No parameters.

See:

- wolfSSLv3_server_method
- wolfTLSv1_server_method
- wolfTLSv1_2_server_method
- wolfTLSv1_3_server_method
- wolfDTLSv1_server_method
- wolfSSLv23_server_method
- wolfSSL_CTX_new

Return:

- – If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_1_server_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.58.4.8 function wolfTLSv1_1_client_method

```
WOLFSSL_METHOD * wolfTLSv1_1_client_method(
    void
)
```

The wolfTLSv1_1_client_method() function is used to indicate that the application is a client and will only support the TLS 1.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Parameters:

- **none** No parameters.

See:

- wolfSSLv3_client_method
- wolfTLSv1_client_method
- wolfTLSv1_2_client_method
- wolfTLSv1_3_client_method
- wolfDTLSv1_client_method
- wolfSSLv23_client_method
- wolfSSL_CTX_new

Return:

- – If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_1_client_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.58.4.9 function wolfTLSv1_2_server_method

```
WOLFSSL_METHOD * wolfTLSv1_2_server_method(
    void
)
```

The wolfTLSv1_2_server_method() function is used to indicate that the application is a server and will only support the TLS 1.2 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Parameters:

- **none** No parameters.

See:

- wolfSSLv3_server_method
- wolfTLSv1_server_method
- wolfTLSv1_1_server_method
- wolfTLSv1_3_server_method
- wolfDTLSv1_server_method
- wolfSSLv23_server_method
- wolfSSL_CTX_new

Return:

- – If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_2_server_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.58.4.10 function wolfTLSv1_2_client_method

```
WOLFSSL_METHOD * wolfTLSv1_2_client_method(
    void
)
```

The wolfTLSv1_2_client_method() function is used to indicate that the application is a client and will only support the TLS 1.2 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Parameters:

- **none** No parameters.

See:

- wolfSSLv3_client_method
- wolfTLSv1_client_method
- wolfTLSv1_1_client_method
- wolfTLSv1_3_client_method
- wolfDTLSv1_client_method
- wolfSSLv23_client_method
- wolfSSL_CTX_new

Return:

- – If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_2_client_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.58.4.11 function wolfDTLSv1_client_method

```
WOLFSSL_METHOD * wolfDTLSv1_client_method(
    void
)
```

The wolfDTLSv1_client_method() function is used to indicate that the application is a client and will only support the DTLS 1.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). This function is only available when wolfSSL has been compiled with DTLS support (–enable_dtls, or by defining wolfSSL_DTLS).

Parameters:

- **none** No parameters.

See:

- wolfSSLv3_client_method
- wolfTLSv1_client_method
- wolfTLSv1_1_client_method
- wolfTLSv1_2_client_method
- wolfTLSv1_3_client_method
- wolfSSLv23_client_method
- wolfSSL_CTX_new

Return:

- – If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.

- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLSv1_client_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.58.4.12 function wolfDTLSv1_server_method

```
WOLFSSL_METHOD * wolfDTLSv1_server_method(
    void
)
```

The wolfDTLSv1_server_method() function is used to indicate that the application is a server and will only support the DTLS 1.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). This function is only available when wolfSSL has been compiled with DTLS support (–enable_dtls, or by defining wolfSSL_DTLS).

Parameters:

- **none** No parameters.

See:

- wolfSSLv3_server_method
- wolfTLSv1_server_method
- wolfTLSv1_1_server_method
- wolfTLSv1_2_server_method
- wolfTLSv1_3_server_method
- wolfSSLv23_server_method
- wolfSSL_CTX_new

Return:

- – If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLSv1_server_method();
if (method == NULL) {
    // unable to get method
}
```

```
ctx = wolfSSL_CTX_new(method);
...
```

C.58.4.13 function wolfDTLSv1_3_server_method

```
WOLFSSL_METHOD * wolfDTLSv1_3_server_method(
    void
)
```

The wolfDTLSv1_3_server_method() function is used to indicate that the application is a server and will only support the DTLS 1.3 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). This function is only available when wolfSSL has been compiled with DTLSv1.3 support (–enable_dtls13, or by defining wolfSSL_DTLS13).

Parameters:

- **none** No parameters.

See: wolfDTLSv1_3_client_method

Return:

- – If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLSv1_3_server_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.58.4.14 function wolfDTLSv1_3_client_method

```
WOLFSSL_METHOD * wolfDTLSv1_3_client_method(
    void
)
```

The wolfDTLSv1_3_client_method() function is used to indicate that the application is a client and will only support the DTLS 1.3 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). This function is only available when wolfSSL has been compiled with DTLSv1.3 support (–enable_dtls13, or by defining wolfSSL_DTLS13).

Parameters:

- **none** No parameters.

See: wolfDTLSv1_3_server_method

Return:

- – If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.

- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLSv1_3_client_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.58.4.15 function wolfDTLS_server_method

```
WOLFSSL_METHOD * wolfDTLS_server_method(
    void
)
```

The wolfDTLS_server_method() function is used to indicate that the application is a server and will support the highest version of DTLS available and all the version up to the minimum version allowed. The default minimum version allowed is based on the define WOLFSSL_MIN_DTLS_DOWNGRADE and can be changed at runtime using wolfSSL_SetMinVersion(). This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). This function is only available when wolfSSL has been compiled with DTLS support (–enable_dtls, or by defining wolfSSL_DTLS).

Parameters:

- **none** No parameters.

See:

- wolfDTLS_client_method
- wolfSSL_SetMinVersion

Return:

- – If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLS_server_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.58.4.16 function wolfDTLS_client_method

```
WOLFSSL_METHOD * wolfDTLS_client_method(
    void
)
```

The wolfDTLS_client_method() function is used to indicate that the application is a client and will support the highest version of DTLS available and all the version up to the minimum version allowed. The default minimum version allowed is based on the define WOLFSSL_MIN_DTLS_DOWNGRADE and can be changed at runtime using wolfSSL_SetMinVersion(). This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). This function is only available when wolfSSL has been compiled with DTLS support (–enable_dtls, or by defining wolfSSL_DTLS).

Parameters:

- **none** No parameters.

See:

- wolfDTLS_server_method
- wolfSSL_SetMinVersion

Return:

- – If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLS_client_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.58.4.17 function wolfDTLSv1_2_server_method

```
WOLFSSL_METHOD * wolfDTLSv1_2_server_method(
    void
)
```

This function creates and initializes a WOLFSSL_METHOD for the server side.

Parameters:

- **none** No parameters.

See: wolfSSL_CTX_new

Return: This function returns a WOLFSSL_METHOD pointer.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(wolfDTLSv1_2_server_method());
WOLFSSL* ssl = WOLFSSL_new(ctx);
…
```

C.58.4.18 function wolfSSL_use_old_poly

```
int wolfSSL_use_old_poly(
    WOLFSSL * ssl,
    int value
)
```

Since there is some differences between the first release and newer versions of chacha-poly AEAD construction we have added an option to communicate with servers/clients using the older version. By default wolfSSL uses the new version.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **value** whether or not to use the older version of setting up the information for poly1305. Passing a flag value of 1 indicates yes use the old poly AEAD, to switch back to using the new version pass a flag value of 0.

See: none

Return: 0 upon success

Example

```
int ret = 0;
WOLFSSL* ssl;
...

ret = wolfSSL_use_old_poly(ssl, 1);
if (ret != 0) {
    // failed to set poly1305 AEAD version
}
```

C.58.4.19 function wolfSSL_dtls_import

```
int wolfSSL_dtls_import(
    WOLFSSL * ssl,
    const unsigned char * buf,
    unsigned int sz
)
```

The wolfSSL_dtls_import() function is used to parse in a serialized session state. This allows for picking up the connection after the handshake has been completed. Post-handshake authentication is not available on an imported connection.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **buf** serialized session to import.
- **sz** size of serialized session buffer.

See:

- wolfSSL_new
- wolfSSL_CTX_new
- wolfSSL_CTX_dtls_set_export

Return:

- Success If successful, the amount of the buffer read will be returned.
- Failure All unsuccessful return values will be less than 0.
- VERSION_ERROR If a version mismatch is found ie DTLS v1 and ctx was set up for DTLS v1.2 then VERSION_ERROR is returned.

Example

```
WOLFSSL* ssl;
int ret;
unsigned char buf[MAX];
bufSz = MAX;
...
//get information sent from wc_dtls_export function and place it in buf
fread(buf, 1, bufSz, input);
ret = wolfSSL_dtls_import(ssl, buf, bufSz);
if (ret < 0) {
// handle error case
}
// no wolfSSL_accept needed since handshake was already done
...
ret = wolfSSL_write(ssl) and wolfSSL_read(ssl);
...
```

C.58.4.20 function wolfSSL_tls_import

```
int wolfSSL_tls_import(
    WOLFSSL * ssl,
    const unsigned char * buf,
    unsigned int sz
)
```

Used to import a serialized TLS session. This function is for importing the state of the connection. A TLS 1.3 session serialized with export version 7 or later carries its traffic and resumption secrets, so a KeyUpdate and the issuing of a NewSessionTicket work on the imported connection; an older serialization restores the record layer alone. WARNING: buf contains sensitive information about the state and is best to be encrypted before storing if stored. Additional debug info can be displayed with the macro WOLFSSL_SESSION_EXPORT_DEBUG defined.

Parameters:

- **ssl** WOLFSSL structure to import the session into
- **buf** serialized session
- **sz** size of buffer 'buf'

See:

- wolfSSL_dtls_import
- wolfSSL_tls_export

Return: the number of bytes read from buffer 'buf'

C.58.4.21 function wolfSSL_CTX_dtls_set_export

```
int wolfSSL_CTX_dtls_set_export(
    WOLFSSL_CTX * ctx,
    wc_dtls_export func
)
```

The wolfSSL_CTX_dtls_set_export() function is used to set the callback function for exporting a session. It is allowed to pass in NULL as the parameter func to clear the export function previously stored. Used on the server side and is called immediately after handshake is completed, for DTLS 1.2 only; a DTLS 1.3 session is exported with wolfSSL_dtls_export().

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().
- **func** wc_dtls_export function to use when exporting a session.

See:

- wolfSSL_new
- wolfSSL_CTX_new
- wolfSSL_dtls_set_export
- Static buffer use

Return:

- SSL_SUCCESS upon success.
- BAD_FUNC_ARG If null or not expected arguments are passed in

Example

```
int send_session(WOLFSSL* ssl, byte* buf, word32 sz, void* userCtx);
// body of send session (wc_dtls_export) that passes
// buf (serialized session) to destination
WOLFSSL_CTX* ctx;
int ret;
...
ret = wolfSSL_CTX_dtls_set_export(ctx, send_session);
if (ret != SSL_SUCCESS) {
    // handle error case
}
...
ret = wolfSSL_accept(ssl);
...
```

C.58.4.22 function wolfSSL_dtls_set_export

```
int wolfSSL_dtls_set_export(
    WOLFSSL * ssl,
    wc_dtls_export func
)
```

The wolfSSL_dtls_set_export() function is used to set the callback function for exporting a session. It is allowed to pass in NULL as the parameter func to clear the export function previously stored. Used on the server side and is called immediately after handshake is completed, for DTLS 1.2 only; a DTLS 1.3 session is exported with wolfSSL_dtls_export().

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **func** wc_dtls_export function to use when exporting a session.

See:

- wolfSSL_new
- wolfSSL_CTX_new
- wolfSSL_CTX_dtls_set_export

Return:

- SSL_SUCCESS upon success.
- BAD_FUNC_ARG If null or not expected arguments are passed in

Example

```
int send_session(WOLFSSL* ssl, byte* buf, word32 sz, void* userCtx);
// body of send session (wc_dtls_export) that passes
// buf (serialized session) to destination
WOLFSSL* ssl;
int ret;
...
ret = wolfSSL_dtls_set_export(ssl, send_session);
if (ret != SSL_SUCCESS) {
    // handle error case
}
...
ret = wolfSSL_accept(ssl);
...
```

C.58.4.23 function wolfSSL_dtls_export

```
int wolfSSL_dtls_export(
    WOLFSSL * ssl,
    unsigned char * buf,
    unsigned int * sz
)
```

The wolfSSL_dtls_export() function is used to serialize a WOLFSSL session into the provided buffer. Allows for less memory overhead than using a function callback for sending a session and choice over when the session is serialized. If buffer is NULL when passed to function then sz will be set to the size of buffer needed for serializing the WOLFSSL session. The export is refused with a negative return before a DTLS 1.3 handshake is done, while a KeyUpdate is in progress, a fragmented message is half sent, a handshake message other than NewSessionTicket is waiting for the peer's ACK, a post-handshake CertificateRequest is unanswered, or a write dup of the object exists. A NewSessionTicket still waiting for its ACK is not exported and is lost. The session tickets themselves are not serialized: any ticket already issued or received is discarded by the export. For a DTLS 1.3 session the blob carries the resumption secret and the ticket nonce, so an imported server can still issue a NewSessionTicket that resumes; use wolfSSL_get1_session before exporting to keep a received ticket.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **buf** buffer to hold serialized session.
- **sz** size of buffer.

See:

- wolfSSL_new
- wolfSSL_CTX_new
- wolfSSL_CTX_dtls_set_export
- wolfSSL_dtls_import

Return:

- Success If successful, the amount of the buffer used will be returned.
- Failure All unsuccessful return values will be less than 0.

Example

```
WOLFSSL* ssl;
int ret;
unsigned char buf[MAX];
bufSz = MAX;
...
ret = wolfSSL_dtls_export(ssl, buf, bufSz);
if (ret < 0) {
    // handle error case
}
...
```

C.58.4.24 function wolfSSL_tls_export

```
int wolfSSL_tls_export(
    WOLFSSL * ssl,
    unsigned char * buf,
    unsigned int * sz
)
```

Used to export a serialized TLS session. This function is for exporting a serialized state of the connection. In most cases wolfSSL_get1_session should be used instead of wolfSSL_tls_export. The export of a TLS 1.3 session is refused with a negative return before the handshake is done or while a KeyUpdate response is pending. The session tickets themselves are not serialized: any ticket already issued or received is discarded by the export. A TLS 1.3 blob carries the resumption secret and the ticket nonce, so an imported server can still issue a NewSessionTicket that resumes; use wolfSSL_get1_session before exporting to keep a received ticket. Additional debug info can be displayed with the macro WOLFSSL_SESSION_EXPORT_DEBUG defined. WARNING: buf contains sensitive information about the state and is best to be encrypted before storing if stored.

Parameters:

- **ssl** WOLFSSL structure to export the session from
- **buf** output of serialized session
- **sz** size in bytes set in 'buf'

See:

- wolfSSL_dtls_import
- wolfSSL_tls_import

Return: the number of bytes written into buffer 'buf'

C.58.4.25 function wolfSSL_CTX_load_static_memory

```
int wolfSSL_CTX_load_static_memory(
    WOLFSSL_CTX ** ctx,
    wolfSSL_method_func method,
    unsigned char * buf,
    unsigned int sz,
    int flag,
    int max
)
```

This function is used to set aside static memory for a CTX. Memory set aside is then used for the CTX's lifetime and for any SSL objects created from the CTX. By passing in a NULL ctx pointer and a wolfSSL_method_func function the creation of the CTX itself will also use static memory. wolfSSL_method_func has the function signature of WOLFSSL_METHOD* (*wolfSSL_method_func)(void* heap);. Passing in 0 for max makes it behave as if not set and no max concurrent use restrictions is

in place. The flag value passed in determines how the memory is used and behavior while operating. Available flags are the following: 0 - default general memory, WOLFMEM_IO_POOL - used for input/output buffer when sending receiving messages and overrides general memory, so all memory in buffer passed in is used for IO, WOLFMEM_IO_FIXED - same as WOLFMEM_IO_POOL but each SSL now keeps two buffers to themselves for their lifetime, WOLFMEM_TRACK_STATS - each SSL keeps track of memory stats while running.

Parameters:

- **ctx** address of pointer to a WOLFSSL_CTX structure.
- **method** function to create protocol. (should be NULL if ctx is not also NULL)
- **buf** memory to use for all operations.
- **sz** size of memory buffer being passed in.
- **flag** type of memory.
- **max** max concurrent operations.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_is_static_memory
- wolfSSL_is_static_memory

Return:

- SSL_SUCCESS upon success.
- SSL_FAILURE upon failure.

Example

```
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
int ret;
unsigned char memory[MAX];
int memorySz = MAX;
unsigned char IO[MAX];
int IOSz = MAX;
int flag = WOLFMEM_IO_FIXED | WOLFMEM_TRACK_STATS;
...
// create ctx also using static memory, start with general memory to use
ctx = NULL:
ret = wolfSSL_CTX_load_static_memory(&ctx, wolfSSLv23_server_method_ex,
memory, memorySz, 0,    MAX_CONCURRENT_HANDSHAKES);
if (ret != SSL_SUCCESS) {
// handle error case
}
// load in memory for use with IO
ret = wolfSSL_CTX_load_static_memory(&ctx, NULL, IO, IOSz, flag,
MAX_CONCURRENT_IO);
if (ret != SSL_SUCCESS) {
// handle error case
}
...
```

C.58.4.26 function wolfSSL_CTX_is_static_memory

```
int wolfSSL_CTX_is_static_memory(
    WOLFSSL_CTX * ctx,
```

```
    WOLFSSL_MEM_STATS * mem_stats
)
```

This function does not change any of the connections behavior and is used only for gathering information about the static memory usage.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **mem_stats** structure to hold information about static memory usage.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_load_static_memory
- wolfSSL_is_static_memory

Return:

- 1 is returned if using static memory for the CTX is true.
- 0 is returned if not using static memory.

Example

```
WOLFSSL_CTX* ctx;
int ret;
WOLFSSL_MEM_STATS mem_stats;
...
//get information about static memory with CTX
ret = wolfSSL_CTX_is_static_memory(ctx, &mem_stats);
if (ret == 1) {
    // handle case of is using static memory
    // print out or inspect elements of mem_stats
}
if (ret == 0) {
    //handle case of ctx not using static memory
}
…
```

C.58.4.27 function wolfSSL_is_static_memory

```
int wolfSSL_is_static_memory(
    WOLFSSL * ssl,
    WOLFSSL_MEM_CONN_STATS * mem_stats
)
```

wolfSSL_is_static_memory is used to gather information about a SSL's static memory usage. The return value indicates if static memory is being used and WOLFSSL_MEM_CONN_STATS will be filled out if and only if the flag WOLFMEM_TRACK_STATS was passed to the parent CTX when loading in static memory.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **mem_stats** structure to contain static memory usage.

See:

- wolfSSL_new
- wolfSSL_CTX_is_static_memory

Return:

- 1 is returned if using static memory for the CTX is true.
- 0 is returned if not using static memory.

Example

```
WOLFSSL* ssl;
int ret;
WOLFSSL_MEM_CONN_STATS mem_stats;
...
ret = wolfSSL_is_static_memory(ssl, mem_stats);
if (ret == 1) {
    // handle case when is static memory
    // investigate elements in mem_stats if WOLFMEM_TRACK_STATS flag
}
...
```

C.58.4.28 function wolfSSL_CTX_use_certificate_file

```
int wolfSSL_CTX_use_certificate_file(
    WOLFSSL_CTX * ctx,
    const char * file,
    int format
)
```

This function loads a certificate file into the SSL context (WOLFSSL_CTX). The file is provided by the file argument. The format argument specifies the format type of the file, either SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. Please see the examples for proper usage.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new()
- **file** a pointer to the name of the file containing the certificate to be loaded into the wolfSSL SSL context.
- **format** - format of the certificates pointed to by file. Possible options are SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_use_certificate_file
- wolfSSL_use_certificate_buffer

Return:

- SSL_SUCCESS upon success.
- SSL_FAILURE If the function call fails, possible causes might include the file is in the wrong format, or the wrong format has been given using the “format” argument, file doesn’t exist, can’t be read, or is corrupted, an out of memory condition occurs, Base16 decoding fails on the file.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_use_certificate_file(ctx, “./client-cert.pem”,
                                       SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading cert file
}
...
```

C.58.4.29 function wolfSSL_CTX_use_PrivateKey_file

```
int wolfSSL_CTX_use_PrivateKey_file(
    WOLFSSL_CTX * ctx,
    const char * file,
    int format
)
```

This function loads a private key file into the SSL context (WOLFSSL_CTX). The file is provided by the file argument. The format argument specifies the format type of the file - SSL_FILETYPE_ASN1or SSL_FILETYPE_PEM. Please see the examples for proper usage.

Parameters:

- **ctx** pointer to a WOLFSSL_CTX structure.
- **file** path to the private key file.
- **format** format of the key file (SSL_FILETYPE_PEM or SSL_FILETYPE_ASN1).

See:

- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_use_PrivateKey_file
- wolfSSL_use_PrivateKey_buffer
- wc_CryptoCb_RegisterDevice
- wolfSSL_CTX_SetDevId

Return:

- SSL_SUCCESS upon success.
- SSL_FAILURE The file is in the wrong format, or the wrong format has been given using the "format" argument. The file doesn't exist, can't be read, or is corrupted. An out of memory condition occurs. Base16 decoding fails on the file. The key file is encrypted but no password is provided.

If using an external key store and do not have the private key you can instead provide the public key and register the crypro callback to handle the signing. For this you can build with either build with crypto callbacks or PK callbacks. To enable crypto callbacks use –enable-cryptocb or WOLF_CRYPTO_CB and register a crypto callback using wc_CryptoCb_RegisterDevice and set the associated devId using wolfSSL_CTX_SetDevId.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_use_PrivateKey_file(ctx, "./server-key.pem",
                                      SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading key file
}
...
```

C.58.4.30 function wolfSSL_CTX_load_verify_locations

```
int wolfSSL_CTX_load_verify_locations(
    WOLFSSL_CTX * ctx,
    const char * file,
    const char * path
)
```

This function loads PEM-formatted CA certificate files into the SSL context (WOLFSSL_CTX). These certificates will be treated as trusted root certificates and used to verify certs received from peers during the SSL handshake. The root certificate file, provided by the file argument, may be a single certificate or a file containing multiple certificates. If multiple CA certs are included in the same file, wolfSSL will load them in the same order they are presented in the file. The path argument is a pointer to the name of a directory that contains certificates of trusted root CAs. If the value of file is not NULL, path may be specified as NULL if not needed. If path is specified and NO_WOLFSSL_DIR was not defined when building the library, wolfSSL will load all CA certificates located in the given directory. This function will attempt to load all files in the directory. This function expects PEM formatted CERT_TYPE file with header "—–BEGIN CERTIFICATE—–".

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **file** pointer to name of the file containing PEM-formatted CA certificates.
- **path** pointer to the name of a directory to load PEM-formatted certificates from.

See:

- wolfSSL_CTX_load_verify_locations_ex
- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_use_certificate_file
- wolfSSL_CTX_use_PrivateKey_file
- wolfSSL_CTX_use_certificate_chain_file
- wolfSSL_use_certificate_file
- wolfSSL_use_PrivateKey_file
- wolfSSL_use_certificate_chain_file

Return:

- SSL_SUCCESS up success.
- SSL_FAILURE will be returned if ctx is NULL, or if both file and path are NULL.
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- ASN_BEFORE_DATE_E will be returned if the current date is before the before date.
- ASN_AFTER_DATE_E will be returned if the current date is after the after date.
- BUFFER_E will be returned if a chain buffer is bigger than the receiving buffer.
- BAD_PATH_ERROR will be returned if opendir() fails when trying to open path.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_load_verify_locations(ctx, "./ca-cert.pem", NULL);
if (ret != WOLFSSL_SUCCESS) {
    // error loading CA certs
}
...
```

C.58.4.31 function wolfSSL_CTX_load_verify_locations_ex

```
int wolfSSL_CTX_load_verify_locations_ex(
    WOLFSSL_CTX * ctx,
    const char * file,
    const char * path,
```

```
    word32 flags
)
```

This function loads PEM-formatted CA certificate files into the SSL context (WOLFSSL_CTX). These certificates will be treated as trusted root certificates and used to verify certs received from peers during the SSL handshake. The root certificate file, provided by the file argument, may be a single certificate or a file containing multiple certificates. If multiple CA certs are included in the same file, wolfSSL will load them in the same order they are presented in the file. The path argument is a pointer to the name of a directory that contains certificates of trusted root CAs. If the value of file is not NULL, path may be specified as NULL if not needed. If path is specified and NO_WOLFSSL_DIR was not defined when building the library, wolfSSL will load all CA certificates located in the given directory. This function will attempt to load all files in the directory based on flags specified. This function expects PEM formatted CERT_TYPE files with header "—–BEGIN CERTIFICATE—–".

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **file** pointer to name of the file containing PEM-formatted CA certificates.
- **path** pointer to the name of a directory to load PEM-formatted certificates from.
- **flags** possible mask values are: WOLFSSL_LOAD_FLAG_IGNORE_ERR, WOLFSSL_LOAD_FLAG_DATE_ERR_OKAY and WOLFSSL_LOAD_FLAG_PEM_CA_ONLY

See:

- wolfSSL_CTX_load_verify_locations
- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_use_certificate_file
- wolfSSL_CTX_use_PrivateKey_file
- wolfSSL_CTX_use_certificate_chain_file
- wolfSSL_use_certificate_file
- wolfSSL_use_PrivateKey_file
- wolfSSL_use_certificate_chain_file

Return:

- SSL_SUCCESS up success.
- SSL_FAILURE will be returned if ctx is NULL, or if both file and path are NULL. This will also be returned if at least one cert is loaded successfully but there is one or more that failed. Check error stack for reason.
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- BUFFER_E will be returned if a chain buffer is bigger than the receiving buffer.
- BAD_PATH_ERROR will be returned if opendir() fails when trying to open path.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_load_verify_locations_ex(ctx, NULL, "./certs/external",
    WOLFSSL_LOAD_FLAG_PEM_CA_ONLY);
if (ret != WOLFSSL_SUCCESS) {
    // error loading CA certs
}
...
```

C.58.4.32 function wolfSSL_get_system_CA_dirs

```
const char ** wolfSSL_get_system_CA_dirs(
    word32 * num
)
```

This function returns a pointer to an array of strings representing directories wolfSSL will search for system CA certs when wolfSSL_CTX_load_system_CA_certs is called. On systems that don't store certificates in an accessible system directory (such as Apple platforms), this function will always return NULL.

Parameters:

- **num** pointer to a word32 that will be populated with the length of the array of strings.

See:

- wolfSSL_CTX_load_system_CA_certs
- wolfSSL_CTX_load_verify_locations
- wolfSSL_CTX_load_verify_locations_ex

Return:

- Valid pointer on success.
- NULL pointer on failure.

Example

```
WOLFSSL_CTX* ctx;
const char** dirs;
word32 numDirs;

dirs = wolfSSL_get_system_CA_dirs(&numDirs);
for (int i = 0; i < numDirs; ++i) {
    printf("Potential system CA dir: %s\n", dirs[i]);
}
...
```

C.58.4.33 function wolfSSL_CTX_load_system_CA_certs

```
int wolfSSL_CTX_load_system_CA_certs(
    WOLFSSL_CTX * ctx
)
```

On most platforms (including Linux and Windows), this function attempts to load CA certificates into a WOLFSSL_CTX from an OS-dependent CA certificate store. Loaded certificates will be trusted.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().

See:

- wolfSSL_get_system_CA_dirs
- wolfSSL_CTX_load_verify_locations
- wolfSSL_CTX_load_verify_locations_ex

Return:

- WOLFSSL_SUCCESS on success.
- WOLFSSL_BAD_PATH if no system CA certs were loaded.
- WOLFSSL_FAILURE for other failure types (e.g. Windows cert store wasn't properly closed).

On Apple platforms (excluding macOS), certificates can't be obtained from the system, and therefore cannot be loaded into the wolfSSL certificate manager. For these platforms, this function enables TLS connections bound to the WOLFSSL_CTX to use the native system trust APIs to verify authenticity of the peer certificate chain if the authenticity of the peer cannot first be authenticated against certificates loaded by the user.

The platforms supported and tested are: Linux (Debian, Ubuntu, Gentoo, Fedora, RHEL), Windows 10/11, Android, macOS, and iOS.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_load_system_CA_certs(ctx,);
if (ret != WOLFSSL_SUCCESS) {
    // error loading system CA certs
}
...
```

C.58.4.34 function wolfSSL_CTX_trust_peer_cert

```
int wolfSSL_CTX_trust_peer_cert(
    WOLFSSL_CTX * ctx,
    const char * file,
    int type
)
```

This function loads a certificate to use for verifying a peer when performing a TLS/SSL handshake. The peer certificate sent during the handshake is compared by using the SKID when available and the signature. If these two things do not match then any loaded CAs are used. Feature is enabled by defining the macro WOLFSSL_TRUST_PEER_CERT. Please see the examples for proper usage.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **file** pointer to name of the file containing certificates
- **type** type of certificate being loaded ie SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_use_certificate_file
- wolfSSL_CTX_use_PrivateKey_file
- wolfSSL_CTX_use_certificate_chain_file
- wolfSSL_CTX_trust_peer_buffer
- wolfSSL_CTX_Unload_trust_peers
- wolfSSL_use_certificate_file
- wolfSSL_use_PrivateKey_file
- wolfSSL_use_certificate_chain_file

Return:

- SSL_SUCCES upon success.
- SSL_FAILURE will be returned if ctx is NULL, or if both file and type are invalid.
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
...

ret = wolfSSL_CTX_trust_peer_cert(ctx, "./peer-cert.pem",
SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading trusted peer cert
}
...
```

C.58.4.35 function wolfSSL_CTX_use_certificate_chain_file

```
int wolfSSL_CTX_use_certificate_chain_file(
    WOLFSSL_CTX * ctx,
    const char * file
)
```

This function loads a chain of certificates into the SSL context (WOLFSSL_CTX). The file containing the certificate chain is provided by the file argument, and must contain PEM-formatted certificates. This function will process up to MAX_CHAIN_DEPTH (default = 9, defined in internal.h) certificates, plus the subject cert.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new()
- **file** a pointer to the name of the file containing the chain of certificates to be loaded into the wolfSSL SSL context. Certificates must be in PEM format.

See:

- wolfSSL_CTX_use_certificate_file
- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_use_certificate_file
- wolfSSL_use_certificate_buffer

Return:

- SSL_SUCCESS upon success
- SSL_FAILURE If the function call fails, possible causes might include the file is in the wrong format, or the wrong format has been given using the "format" argument, file doesn't exist, can't be read, or is corrupted, an out of memory condition occurs.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_use_certificate_chain_file(ctx, "./cert-chain.pem");
if (ret != SSL_SUCCESS) {
    // error loading cert file
}
...
```

C.58.4.36 function wolfSSL_CTX_use_RSAPrivateKey_file

```
int wolfSSL_CTX_use_RSAPrivateKey_file(
    WOLFSSL_CTX * ctx,
    const char * file,
    int format
)
```

This function loads the private RSA key used in the SSL connection into the SSL context (WOLFSSL_CTX). This function is only available when wolfSSL has been compiled with the OpenSSL compatibility layer enabled (–enable-opensslExtra, #define OPENSSL_EXTRA), and is identical to the more-typically used wolfSSL_CTX_use_PrivateKey_file() function. The file argument contains a pointer to the RSA private key file, in the format specified by format.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new()
- **file** a pointer to the name of the file containing the RSA private key to be loaded into the wolfSSL SSL context, with format as specified by format.
- **format** the encoding type of the RSA private key specified by file. Possible values include SSL_FILETYPE_PEM and SSL_FILETYPE_ASN1.

See:

- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_PrivateKey_file
- wolfSSL_use_RSAPrivateKey_file
- wolfSSL_use_PrivateKey_buffer
- wolfSSL_use_PrivateKey_file

Return:

- SSL_SUCCESS upon success.
- SSL_FAILURE If the function call fails, possible causes might include: The input key file is in the wrong format, or the wrong format has been given using the “format” argument, file doesn’t exist, can’t be read, or is corrupted, an out of memory condition occurs.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_use_RSAPrivateKey_file(ctx, “./server-key.pem”,
                                         SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading private key file
}
...
```

C.58.4.37 function wolfSSL_get_verify_depth

```
long wolfSSL_get_verify_depth(
    WOLFSSL * ssl
)
```

This function returns the maximum chain depth allowed, which is 9 by default, for a valid session i.e. there is a non-null session object (ssl).

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_CTX_get_verify_depth

Return:

- MAX_CHAIN_DEPTH returned if the WOLFSSL structure is not NULL. By default the value is 9.
- BAD_FUNC_ARG returned if the WOLFSSL structure is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
long sslDep = wolfSSL_get_verify_depth(ssl);

if(sslDep > EXPECTED){
    // The verified depth is greater than what was expected
} else {
    // The verified depth is smaller or equal to the expected value
}
```

C.58.4.38 function wolfSSL_CTX_get_verify_depth

```
long wolfSSL_CTX_get_verify_depth(
    WOLFSSL_CTX * ctx
)
```

This function gets the certificate chaining depth using the CTX structure.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

See:

- wolfSSL_CTX_use_certificate_chain_file
- wolfSSL_get_verify_depth

Return:

- MAX_CHAIN_DEPTH returned if the CTX struct is not NULL. The constant representation of the max certificate chain peer depth.
- BAD_FUNC_ARG returned if the CTX structure is NULL.

Example

```
WOLFSSL_METHOD method; // protocol method
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(method);
…
long ret = wolfSSL_CTX_get_verify_depth(ctx);

if(ret == EXPECTED){
    //  You have the expected value
} else {
    //  Handle an unexpected depth
}
```

C.58.4.39 function wolfSSL_use_certificate_file

```
int wolfSSL_use_certificate_file(
    WOLFSSL * ssl,
    const char * file,
```

```
    int format
)
```

This function loads a certificate file into the SSL session (WOLFSSL structure). The certificate file is provided by the file argument. The format argument specifies the format type of the file - either SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created with wolfSSL_new().
- **file** a pointer to the name of the file containing the certificate to be loaded into the wolfSSL SSL session, with format as specified by format.
- **format** the encoding type of the certificate specified by file. Possible values include SSL_FILETYPE_PEM and SSL_FILETYPE_ASN1.

See:

- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_CTX_use_certificate_file
- wolfSSL_use_certificate_buffer

Return:

- SSL_SUCCESS upon success
- SSL_FAILURE If the function call fails, possible causes might include: The file is in the wrong format, or the wrong format has been given using the “format” argument, file doesn’t exist, can’t be read, or is corrupted, an out of memory condition occurs, Base16 decoding fails on the file

Example

```
int ret = 0;
WOLFSSL* ssl;
...
ret = wolfSSL_use_certificate_file(ssl, “./client-cert.pem”,
                                SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading cert file
}
...
```

C.58.4.40 function wolfSSL_use_PrivateKey_file

```
int wolfSSL_use_PrivateKey_file(
    WOLFSSL * ssl,
    const char * file,
    int format
)
```

This function loads a private key file into the SSL session (WOLFSSL structure). The key file is provided by the file argument. The format argument specifies the format type of the file - SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created with wolfSSL_new().
- **file** a pointer to the name of the file containing the key file to be loaded into the wolfSSL SSL session, with format as specified by format.
- **format** the encoding type of the key specified by file. Possible values include SSL_FILETYPE_PEM and SSL_FILETYPE_ASN1.

See:

- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_PrivateKey_file
- wolfSSL_use_PrivateKey_buffer
- wc_CryptoCb_RegisterDevice
- wolfSSL_SetDevId

Return:

- SSL_SUCCESS upon success.
- SSL_FAILURE If the function call fails, possible causes might include: The file is in the wrong format, or the wrong format has been given using the "format" argument, The file doesn't exist, can't be read, or is corrupted, An out of memory condition occurs, Base16 decoding fails on the file, The key file is encrypted but no password is provided

If using an external key store and do not have the private key you can instead provide the public key and register the crypro callback to handle the signing. For this you can build with either build with crypto callbacks or PK callbacks. To enable crypto callbacks use –enable-cryptocb or WOLF_CRYPTO_CB and register a crypto callback using wc_CryptoCb_RegisterDevice and set the associated devId using wolfSSL_SetDevId.

Example

```
int ret = 0;
WOLFSSL* ssl;
...
ret = wolfSSL_use_PrivateKey_file(ssl, "./server-key.pem",
                                  SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading key file
}
...
```

C.58.4.41 function wolfSSL_use_certificate_chain_file

```
int wolfSSL_use_certificate_chain_file(
    WOLFSSL * ssl,
    const char * file
)
```

This function loads a chain of certificates into the SSL session (WOLFSSL structure). The file containing the certificate chain is provided by the file argument, and must contain PEM-formatted certificates. This function will process up to MAX_CHAIN_DEPTH (default = 9, defined in internal.h) certificates, plus the subject certificate.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new()
- **file** a pointer to the name of the file containing the chain of certificates to be loaded into the wolfSSL SSL session. Certificates must be in PEM format.

See:

- wolfSSL_CTX_use_certificate_chain_file
- wolfSSL_CTX_use_certificate_chain_buffer
- wolfSSL_use_certificate_chain_buffer

Return:

- SSL_SUCCESS upon success.

- SSL_FAILURE If the function call fails, possible causes might include: The file is in the wrong format, or the wrong format has been given using the “format” argument, file doesn’t exist, can’t be read, or is corrupted, an out of memory condition occurs

Example

```
int ret = 0;
WOLFSSL* ctx;
...
ret = wolfSSL_use_certificate_chain_file(ssl, “./cert-chain.pem”);
if (ret != SSL_SUCCESS) {
    // error loading cert file
}
...
```

C.58.4.42 function wolfSSL_use_RSAPrivateKey_file

```
int wolfSSL_use_RSAPrivateKey_file(
    WOLFSSL * ssl,
    const char * file,
    int format
)
```

This function loads the private RSA key used in the SSL connection into the SSL session (WOLFSSL structure). This function is only available when wolfSSL has been compiled with the OpenSSL compatibility layer enabled (–enable-opensslExtra, #define OPENSSL_EXTRA), and is identical to the more-typically used wolfSSL_use_PrivateKey_file() function. The file argument contains a pointer to the RSA private key file, in the format specified by format.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new()
- **file** a pointer to the name of the file containing the RSA private key to be loaded into the wolfSSL SSL session, with format as specified by format.
- **format** the encoding type of the RSA private key specified by file. Possible values include SSL_FILETYPE_PEM and SSL_FILETYPE_ASN1.

See:

- wolfSSL_CTX_use_RSAPrivateKey_file
- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_PrivateKey_file
- wolfSSL_use_PrivateKey_buffer
- wolfSSL_use_PrivateKey_file

Return:

- SSL_SUCCESS upon success
- SSL_FAILURE If the function call fails, possible causes might include: The input key file is in the wrong format, or the wrong format has been given using the “format” argument, file doesn’t exist, can’t be read, or is corrupted, an out of memory condition occurs

Example

```
int ret = 0;
WOLFSSL* ssl;
...
ret = wolfSSL_use_RSAPrivateKey_file(ssl, “./server-key.pem”,
                                     SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
```

```
    // error loading private key file
}
...
```

C.58.4.43 function wolfSSL_CTX_der_load_verify_locations

```
int wolfSSL_CTX_der_load_verify_locations(
    WOLFSSL_CTX * ctx,
    const char * file,
    int format
)
```

This function is similar to wolfSSL_CTX_load_verify_locations, but allows the loading of DER-formatted CA files into the SSL context (WOLFSSL_CTX). It may still be used to load PEM-formatted CA files as well. These certificates will be treated as trusted root certificates and used to verify certs received from peers during the SSL handshake. The root certificate file, provided by the file argument, may be a single certificate or a file containing multiple certificates. If multiple CA certs are included in the same file, wolfSSL will load them in the same order they are presented in the file. The format argument specifies the format which the certificates are in either, SSL_FILETYPE_PEM or SSL_FILETYPE_ASN1 (DER). Unlike wolfSSL_CTX_load_verify_locations, this function does not allow the loading of CA certificates from a given directory path. Note that this function is only available when the wolfSSL library was compiled with WOLFSSL_DER_LOAD defined.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new()
- **file** a pointer to the name of the file containing the CA certificates to be loaded into the wolfSSL SSL context, with format as specified by format.
- **format** the encoding type of the certificates specified by file. Possible values include SSL_FILETYPE_PEM and SSL_FILETYPE_ASN1.

See:

- wolfSSL_CTX_load_verify_locations
- wolfSSL_CTX_load_verify_buffer

Return:

- SSL_SUCCESS upon success.
- SSL_FAILURE upon failure.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_der_load_verify_locations(ctx, "./ca-cert.der",
                                            SSL_FILETYPE_ASN1);
if (ret != SSL_SUCCESS) {
    // error loading CA certs
}
...
```

C.58.4.44 function wolfSSL_CTX_new

```
WOLFSSL_CTX * wolfSSL_CTX_new(
    WOLFSSL_METHOD *
)
```

This function creates a new SSL context, taking a desired SSL/TLS protocol method for input.

Parameters:

- **method** pointer to the desired WOLFSSL_METHOD to use for the SSL context. This is created using one of the wolfSSLvXX_XXXX_method() functions to specify SSL/TLS/DTLS protocol level. This function frees the passed in WOLFSSL_METHOD struct on failure.

See: wolfSSL_new

Return:

- pointer If successful the call will return a pointer to the newly-created WOLFSSL_CTX.
- NULL upon failure.

Example

```
WOLFSSL_CTX*    ctx    = 0;
WOLFSSL_METHOD* method = 0;

method = wolfSSLv3_client_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // context creation failed
}
```

C.58.4.45 function wolfSSL_new

```
WOLFSSL * wolfSSL_new(
    WOLFSSL_CTX *
)
```

This function creates a new SSL session, taking an already created SSL context as input.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().

See: wolfSSL_CTX_new

Return:

- – If successful the call will return a pointer to the newly-created wolfSSL structure.
- NULL Upon failure.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL*     ssl = NULL;
WOLFSSL_CTX* ctx = 0;

ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // context creation failed
}
```

```
ssl = wolfSSL_new(ctx);
if (ssl == NULL) {
    // SSL object creation failed
}
```

C.58.4.46 function wolfSSL_set_fd

```
int wolfSSL_set_fd(
    WOLFSSL * ssl,
    int fd
)
```

This function assigns a file descriptor (fd) as the input/output facility for the SSL connection. Typically this will be a socket file descriptor.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **fd** file descriptor to use with SSL/TLS connection.

See:

- wolfSSL_CTX_SetIOSend
- wolfSSL_CTX_SetIORecv
- wolfSSL_SetIOReadCtx
- wolfSSL_SetIOWriteCtx

Return:

- SSL_SUCCESS upon success.
- BAD_FUNC_ARG upon failure.

Example

```
int sockfd;
WOLFSSL* ssl = 0;
...

ret = wolfSSL_set_fd(ssl, sockfd);
if (ret != SSL_SUCCESS) {
    // failed to set SSL file descriptor
}
```

C.58.4.47 function wolfSSL_set_dtls_fd_connected

```
int wolfSSL_set_dtls_fd_connected(
    WOLFSSL * ssl,
    int fd
)
```

This function assigns a file descriptor (fd) as the input/output facility for the SSL connection. Typically this will be a socket file descriptor. This is a DTLS specific API because it marks that the socket is connected. recvfrom and sendto calls on this fd will have the addr and addr_len parameters set to NULL.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **fd** file descriptor to use with SSL/TLS connection.

See:

- wolfSSL_CTX_SetIOSend
- wolfSSL_CTX_SetIORecv
- wolfSSL_SetIOReadCtx
- wolfSSL_SetIOWriteCtx
- wolfDTLS_SetChGoodCb

Return:

- SSL_SUCCESS upon success.
- BAD_FUNC_ARG upon failure.

Example

```
int sockfd;
WOLFSSL* ssl = 0;
...
if (connect(sockfd, peer_addr, peer_addr_len) != 0) {
    // handle connect error
}
...
ret = wolfSSL_set_dtls_fd_connected(ssl, sockfd);
if (ret != SSL_SUCCESS) {
    // failed to set SSL file descriptor
}
```

C.58.4.48 function wolfDTLS_SetChGoodCb

```
int wolfDTLS_SetChGoodCb(
    WOLFSSL * ssl,
    ClientHelloGoodCb cb,
    void * user_ctx
)
```

Allows setting a callback for a correctly processed and verified DTLS client hello. When using a cookie exchange mechanism (either the HelloVerifyRequest in DTLS 1.2 or the HelloRetryRequest with a cookie extension in DTLS 1.3) this callback is called after the cookie exchange has succeeded. This is useful to use one WOLFSSL object as the listener for new connections and being able to isolate the WOLFSSL object once the ClientHello is verified (either through a cookie exchange or just checking if the ClientHello had the correct format). With DTLS cookies disabled, the callback is invoked once per ClientHello, only after complete successful ClientHello processing, not on its first fragment. Retries, retransmitted fragments and a second ClientHello after a key-share HelloRetryRequest do not repeat the notification. The object is already stateful before input is read, but the peer's return-routability has not been verified. Returning 0 accepts the ClientHello and the handshake continues. Any negative return stops the accept call with that error and leaves the accept state machine at the ClientHello, so nothing is sent to the peer and the next accept call asks the callback again instead of continuing the handshake. All negative returns behave this way; the value only decides what wolfSSL_get_error() reports, so what it means is the application's choice. A callback that is not ready to decide can return WANT_READ or WANT_WRITE and be asked again as many times as it needs, and one that refuses the peer returns its own error and keeps the server silent by refusing again on every later accept call. The cookie mode cannot be changed from the callback; see wolfSSL_disable_cookie(). DTLS 1.2: https://datatracker.ietf.org/doc/html/rfc6347#section_4.2.1 DTLS 1.3: https://www.rfc_editor.org/rfc/rfc8446#section_4.2.2.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **cb** ClientHelloGoodCb callback function pointer.
- **user_ctx** pointer to user context to be passed to callback.

See: wolfSSL_set_dtls_fd_connected

Return:

- SSL_SUCCESS upon success.
- BAD_FUNC_ARG upon failure.

Example

```
// Called when we have verified a connection
static int chGoodCb(WOLFSSL* ssl, void* arg)
{
    // setup peer and file descriptors

}

if (wolfDTLS_SetChGoodCb(ssl, chGoodCb, NULL) != WOLFSSL_SUCCESS) {
     // error setting callback
}
```

C.58.4.49 function wolfSSL_get_cipher_list

```
char * wolfSSL_get_cipher_list(
    int priority
)
```

Get the name of cipher at priority level passed in.

Parameters:

- **priority** Integer representing the priority level of a cipher.

See:

- wolfSSL_CIPHER_get_name
- wolfSSL_get_current_cipher

Return:

- string Success
- 0 Priority is either out of bounds or not valid.

Example

```
printf("The cipher at 1 is %s", wolfSSL_get_cipher_list(1));
```

C.58.4.50 function wolfSSL_get_ciphers

```
int wolfSSL_get_ciphers(
    char * buf,
    int len
)
```

This function gets the ciphers enabled in wolfSSL.

Parameters:

- **buf** a char pointer representing the buffer.
- **len** the length of the buffer.

See:

- GetCipherNames

- wolfSSL_get_cipher_list
- ShowCiphers

Return:

- SSL_SUCCESS returned if the function executed without error.
- BAD_FUNC_ARG returned if the buf parameter was NULL or if the len argument was less than or equal to zero.
- BUFFER_E returned if the buffer is not large enough and will overflow.

Example

```
static void ShowCiphers(void){
char* ciphers;
int ret = wolfSSL_get_ciphers(ciphers, (int)sizeof(ciphers));

if(ret == SSL_SUCCES){
        printf("%s\n", ciphers);
    }
}
```

C.58.4.51 function wolfSSL_get_cipher_name

```
const char * wolfSSL_get_cipher_name(
    WOLFSSL * ssl
)
```

This function gets the cipher name in the format DHE-RSA by passing through argument to wolfSSL_get_cipher_name_internal.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_CIPHER_get_name
- wolfSSL_get_current_cipher
- wolfSSL_get_cipher_name_internal

Return:

- string This function returns the string representation of the cipher suite that was matched.
- NULL error or cipher not found.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
…
char* cipherS = wolfSSL_get_cipher_name(ssl);

if(cipher == NULL){
    // There was not a cipher suite matched
} else {
    // There was a cipher suite matched
    printf("%s\n", cipherS);
}
```

C.58.4.52 function wolfSSL_get_fd

```
int wolfSSL_get_fd(
    const WOLFSSL * ssl
)
```

This function returns the read file descriptor (fd) used as the input facility for the SSL connection. Typically this will be a socket file descriptor.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See:

- wolfSSL_set_fd
- wolfSSL_set_read_fd
- wolfSSL_set_write_fd

Return: fd If successful the call will return the SSL session file descriptor.

Example

```
int sockfd;
WOLFSSL* ssl = 0;
...
sockfd = wolfSSL_get_fd(ssl);
...
```

C.58.4.53 function wolfSSL_get_wfd

```
int wolfSSL_get_wfd(
    const WOLFSSL * ssl
)
```

This function returns the write file descriptor (fd) used as the output facility for the SSL connection. Typically this will be a socket file descriptor.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See:

- wolfSSL_set_fd
- wolfSSL_set_read_fd
- wolfSSL_set_write_fd

Return: fd If successful the call will return the SSL session file descriptor.

Example

```
int sockfd;
WOLFSSL* ssl = 0;
...
sockfd = wolfSSL_get_wfd(ssl);
...
```

C.58.4.54 function wolfSSL_set_using_nonblock

```
void wolfSSL_set_using_nonblock(
    WOLFSSL * ssl,
```

```
    int nonblock
)
```

This function informs the WOLFSSL object that the underlying I/O is non-blocking. After an application creates a WOLFSSL object, if it will be used with a non-blocking socket, call wolfSSL_set_using_nonblock() on it. This lets the WOLFSSL object know that receiving EWOULDBLOCK means that the recvfrom call would block rather than that it timed out.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **nonblock** value used to set non-blocking flag on WOLFSSL object. Use 1 to specify non-blocking, otherwise 0.

See:

- wolfSSL_get_using_nonblock
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls_get_current_timeout

Return: none No return.

Example

```
WOLFSSL* ssl = 0;
...
wolfSSL_set_using_nonblock(ssl, 1);
```

C.58.4.55 function wolfSSL_get_using_nonblock

```
int wolfSSL_get_using_nonblock(
    WOLFSSL *
)
```

This function allows the application to determine if wolfSSL is using non-blocking I/O. If wolfSSL is using non-blocking I/O, this function will return 1, otherwise 0. After an application creates a WOLFSSL object, if it will be used with a non-blocking socket, call wolfSSL_set_using_nonblock() on it. This lets the WOLFSSL object know that receiving EWOULDBLOCK means that the recvfrom call would block rather than that it timed out.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See: wolfSSL_set_session

Return:

- 0 underlying I/O is blocking.
- 1 underlying I/O is non-blocking.

Example

```
int ret = 0;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_get_using_nonblock(ssl);
if (ret == 1) {
    // underlying I/O is non-blocking
}
...
```

C.58.4.56 function wolfSSL_write

```
int wolfSSL_write(
    WOLFSSL * ssl,
    const void * data,
    int sz
)
```

This function writes sz bytes from the buffer, data, to the SSL connection, ssl. If necessary, wolfSSL_write() will negotiate an SSL/TLS session if the handshake has not already been performed yet by wolfSSL_connect() or wolfSSL_accept(). When using (D)TLSv1.3 and early data feature is compiled in, this function progresses the handshake only up to the point when it is possible to send data. Next invocations of wolfSSL_Connect()/wolfSSL_Accept()/wolfSSL_read() will complete the handshake. wolfSSL_write() works with both blocking and non_blocking I/O. When the underlying I/O is non_blocking, wolfSSL_write() will return when the underlying I/O could not satisfy the needs of wolfSSL_write() to continue. In this case, a call to wolfSSL_get_error() will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_write() when the underlying I/O is ready. If the underlying I/O is blocking, wolfSSL_write() will only return once the buffer data of size sz has been completely written or an error occurred.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **data** data buffer which will be sent to peer.
- **sz** size, in bytes, of data to send to the peer (data).

See:

- wolfSSL_send
- wolfSSL_read
- wolfSSL_recv

Return:

- 0 the number of bytes written upon success.
- 0 will be returned upon failure. Call wolfSSL_get_error() for the specific error code.
- SSL_FATAL_ERROR will be returned upon failure when either an error occurred or, when using non-blocking sockets, the SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE error was received and and the application needs to call wolfSSL_write() again. Use wolfSSL_get_error() to get a specific error code.
- BAD_FUNC_ARG will be returned when ssl or data is NULL, or sz is negative.

Example

```
WOLFSSL* ssl = 0;
char msg[64] = “hello wolfssl!”;
int msgSz = (int)strlen(msg);
int flags;
int ret;
...

ret = wolfSSL_write(ssl, msg, msgSz);
if (ret <= 0) {
    // wolfSSL_write() failed, call wolfSSL_get_error()
}
```

C.58.4.57 function wolfSSL_write_ex

```
int wolfSSL_write_ex(
    WOLFSSL * ssl,
    const void * data,
    size_t sz,
    size_t * wr
)
```

This function writes sz bytes from the buffer, data, to the SSL connection, ssl, and reports the number of bytes written. It is equivalent to wolfSSL_write() except that the length written is returned through wr and the return value only indicates success or failure. Whether a partial write counts as success depends on WOLFSSL_MODE_ENABLE_PARTIAL_WRITE having been set with wolfSSL_CTX_set_mode(); without it, anything short of the full length is a failure.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **data** data buffer to write to the SSL connection.
- **sz** number of bytes to write.
- **wr** pointer that receives the number of bytes written. May be NULL. Set to zero before anything else is done, so it reads as zero even when the call fails. This differs from wolfSSL_read_ex(), which leaves its count alone unless data was read.

See:

- wolfSSL_write
- wolfSSL_read_ex
- wolfSSL_get_error

Return:

- 1 on success.
- 0 on failure. Call wolfSSL_get_error() for the reason.
- BAD_FUNC_ARG when ssl is NULL.

Example

```
WOLFSSL* ssl = 0;
char msg[] = "hello wolfssl!";
size_t written = 0;
...

if (wolfSSL_write_ex(ssl, msg, sizeof(msg), &written) != 1) {
    // handle the failure, see wolfSSL_get_error()
}
```

C.58.4.58 function wolfSSL_read

```
int wolfSSL_read(
    WOLFSSL * ssl,
    void * data,
    int sz
)
```

This function reads sz bytes from the SSL session (ssl) internal read buffer into the buffer data. The bytes read are removed from the internal receive buffer. If necessary wolfSSL_read() will negotiate an SSL/TLS session if the handshake has not already been performed yet by wolfSSL_connect() or wolfSSL_accept(). The SSL/TLS protocol uses SSL records which have a maximum size of 16kB (the max

record size can be controlled by the MAX_RECORD_SIZE define in /wolfssl/internal.h). As such, wolfSSL needs to read an entire SSL record internally before it is able to process and decrypt the record. Because of this, a call to wolfSSL_read() will only be able to return the maximum buffer size which has been decrypted at the time of calling. There may be additional not_yet_decrypted data waiting in the internal wolfSSL receive buffer which will be retrieved and decrypted with the next call to wolfSSL_read(). If sz is larger than the number of bytes in the internal read buffer, SSL_read() will return the bytes available in the internal read buffer. If no bytes are buffered in the internal read buffer yet, a call to wolfSSL_read() will trigger processing of the next record.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **data** buffer where wolfSSL_read() will place data read.
- **sz** number of bytes to read into data.

See:

- wolfSSL_recv
- wolfSSL_write
- wolfSSL_peek
- wolfSSL_pending

Return:

- 0 the number of bytes read upon success.
- 0 will be returned upon failure. This may be caused by a either a clean (close notify alert) shutdown or just that the peer closed the connection. Call wolfSSL_get_error() for the specific error code.
- SSL_FATAL_ERROR will be returned upon failure when either an error occurred or, when using non-blocking sockets, the SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE error was received and and the application needs to call wolfSSL_read() again. Use wolfSSL_get_error() to get a specific error code.
- BAD_FUNC_ARG will be returned when ssl or data is NULL, or sz is negative.

Example

```
WOLFSSL* ssl = 0;
char reply[1024];
...

input = wolfSSL_read(ssl, reply, sizeof(reply));
if (input > 0) {
    // “input” number of bytes returned into buffer “reply”
}

See wolfSSL examples (client, server, echoclient, echoserver) for more
complete examples of wolfSSL_read().
```

C.58.4.59 function wolfSSL_read_ex

```
int wolfSSL_read_ex(
    WOLFSSL * ssl,
    void * data,
    size_t sz,
    size_t * rd
)
```

This function reads up to sz bytes of decrypted application data from the SSL connection, ssl, into the buffer, data, and reports the number of bytes read. It is equivalent to wolfSSL_read() except that the length read is returned through rd and the return value only indicates whether any application data was read.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **data** buffer to hold the data read.
- **sz** size of the buffer in bytes.
- **rd** pointer that receives the number of bytes read. May be NULL and is only set when data was read, so it keeps whatever the caller left in it when the call fails. This differs from wolfSSL_write_ex(), which clears its count first.

See:

- wolfSSL_read
- wolfSSL_write_ex
- wolfSSL_pending
- wolfSSL_get_error

Return:

- 1 when application data was read.
- 0 when no application data was read. Call wolfSSL_get_error() for the reason.
- BAD_FUNC_ARG when ssl is NULL.

Example

```
WOLFSSL* ssl = 0;
char reply[1024];
size_t bytesRead = 0;
...

if (wolfSSL_read_ex(ssl, reply, sizeof(reply), &bytesRead) == 1) {
    // "bytesRead" bytes returned into buffer "reply"
}
```

C.58.4.60 function wolfSSL_peek

```
int wolfSSL_peek(
    WOLFSSL * ssl,
    void * data,
    int sz
)
```

This function copies sz bytes from the SSL session (ssl) internal read buffer into the buffer data. This function is identical to wolfSSL_read() except that the data in the internal SSL session receive buffer is not removed or modified. If necessary, like wolfSSL_read(), wolfSSL_peek() will negotiate an SSL/TLS session if the handshake has not already been performed yet by wolfSSL_connect() or wolfSSL_accept(). The SSL/TLS protocol uses SSL records which have a maximum size of 16kB (the max record size can be controlled by the MAX_RECORD_SIZE define in /wolfssl/internal.h). As such, wolfSSL needs to read an entire SSL record internally before it is able to process and decrypt the record. Because of this, a call to wolfSSL_peek() will only be able to return the maximum buffer size which has been decrypted at the time of calling. There may be additional not_yet_decrypted data waiting in the internal wolfSSL receive buffer which will be retrieved and decrypted with the next call to wolfSSL_peek() / wolfSSL_read(). If sz is larger than the number of bytes in the internal read buffer, SSL_peek() will return the bytes available in the internal read buffer. If no bytes are buffered in the internal read buffer yet, a call to wolfSSL_peek() will trigger processing of the next record.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **data** buffer where wolfSSL_peek() will place data read.
- **sz** number of bytes to read into data.

See: wolfSSL_read

Return:

- 0 the number of bytes read upon success.
- 0 will be returned upon failure. This may be caused by a either a clean (close notify alert) shutdown or just that the peer closed the connection. Call wolfSSL_get_error() for the specific error code.
- SSL_FATAL_ERROR will be returned upon failure when either an error occurred or, when using nonblocking sockets, the SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE error was received and and the application needs to call wolfSSL_peek() again. Use wolfSSL_get_error() to get a specific error code.
- BAD_FUNC_ARG will be returned when ssl or data is NULL, or sz is negative.

Example

```
WOLFSSL* ssl = 0;
char reply[1024];
...

input = wolfSSL_peek(ssl, reply, sizeof(reply));
if (input > 0) {
    // “input” number of bytes returned into buffer “reply”
}
```

C.58.4.61 function wolfSSL_accept

```
int wolfSSL_accept(
    WOLFSSL * ssl
)
```

This function is called on the server side and waits for an SSL client to initiate the SSL/TLS handshake. When this function is called, the underlying communication channel has already been set up. wolfSSL_accept() works with both blocking and non_blocking I/O. When the underlying I/O is non_blocking, wolfSSL_accept() will return when the underlying I/O could not satisfy the needs of wolfSSL_accept to continue the handshake. In this case, a call to wolfSSL_get_error() will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_accept when data is available to read and wolfSSL will pick up where it left off. When using a non_blocking socket, nothing needs to be done, but select() can be used to check for the required condition. If the underlying I/O is blocking, wolfSSL_accept() will only return once the handshake has been finished or an error occurred.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_get_error
- wolfSSL_connect

Return:

- SSL_SUCCESS upon success.
- SSL_FATAL_ERROR will be returned if an error occurred. To get a more detailed error code, call wolfSSL_get_error().

If decrypted application data is waiting to be read, for example early data that has not been read yet or data received during a secure renegotiation, the handshake is not continued and wolfSSL_get_error() returns APP_DATA_READY. Read the data with wolfSSL_read() and then call wolfSSL_accept() again.

Example

```
int ret = 0;
int err = 0;
WOLFSSL* ssl;
char buffer[80];
...

ret = wolfSSL_accept(ssl);
if (ret != SSL_SUCCESS) {
    err = wolfSSL_get_error(ssl, ret);
    printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
}
```

C.58.4.62 function wolfDTLS_accept_stateless

```
int wolfDTLS_accept_stateless(
    WOLFSSL * ssl
)
```

This function statelessly listens for an SSL client to initiate the DTLS handshake. Cookies must be enabled. A cookie-disabled DTLS object is rejected before I/O or callback changes: WOLFSSL_FATAL_ERROR is returned and wolfSSL_get_error() reports BAD_STATE_E. Use wolfSSL_accept() for cookie-disabled connections instead.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_accept
- wolfSSL_get_error
- wolfSSL_connect

Return:

- WOLFSSL_SUCCESS ClientHello containing a valid cookie was received. The connection can be continued with wolfSSL_accept().
- WOLFSSL_FAILURE The I/O layer returned WANT_READ. This is either because there is no data to read and we are using non-blocking sockets or we sent a cookie request and we are waiting for a reply. The user should call wolfDTLS_accept_stateless again after data becomes available in the I/O layer.
- WOLFSSL_FATAL_ERROR A fatal error occurred. The ssl object should be free'd and allocated again to continue. wolfSSL_get_error() reports SIDE_ERROR when called on a client object, and BAD_STATE_E when cookies are disabled on this object.

Example

```
int ret = 0;
int err = 0;
WOLFSSL* ssl;
```

```
...
do {
    ret = wolfDTLS_accept_stateless(ssl);
    if (ret == WOLFSSL_FATAL_ERROR)
        // re-allocate the ssl object with wolfSSL_free() and wolfSSL_new()
} while (ret != WOLFSSL_SUCCESS);
ret = wolfSSL_accept(ssl);
if (ret != SSL_SUCCESS) {
    err = wolfSSL_get_error(ssl, ret);
    printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
}
```

C.58.4.63 function wolfSSL_disable_cookie

```
int wolfSSL_disable_cookie(
    WOLFSSL * ssl
)
```

Disable server cookies for DTLS 1.2, DTLS 1.3 and TLS 1.3, including DTLS 1.2 fallback from DTLS 1.3. Cookies are enabled by default for DTLS. For DTLS, ordinary wolfSSL_accept() or wolfSSL_accept_TLSv13() commits it to stateful processing before the first read, even if a nonblocking accept has no input available. The application must isolate/demultiplex the peer before accepting; disabling cookies removes return_routability verification and exposes the server to DoS/amplification attacks. wolfDTLS_accept_stateless() cannot be used.

Parameters:

- **ssl** DTLS or TLS 1.3 server session created with wolfSSL_new(). A general-purpose session that has not chosen a side yet is accepted: the policy set here is what the promotion to the server side finds.

See:

- wolfSSL_enable_cookie
- wolfDTLS_SetChGoodCb

Return:

- WOLFSSL_SUCCESS on success (including an unchanged mode).
- BAD_FUNC_ARG if ssl is NULL or uses an unsupported protocol (TLS 1.2).
- SIDE_ERROR if ssl is a client.
- BAD_STATE_E if the DTLS handshake already committed to stateful processing.

Primary and secondary cookie secrets for the applicable protocols are securely erased and freed.

Without cookies a DTLS 1.3 server can process a first ClientHello that arrives fragmented, but only when ClientHello fragment reassembly is available: wolfSSL must be built with WOLFSSL_DTLS_CH_FRAG (–enable-dtls-frag-ch, turned on automatically with ML-KEM and DTLS 1.3) and it must be enabled on the object with wolfSSL_dtls13_allow_ch_frag(), which is done by default only in builds with ML-KEM. Otherwise the fragments are dropped and the handshake does not progress.

The DTLS cookie mode is fixed once the handshake commits to stateful processing, which the accept functions do before the first read and stateless processing does when a cookie verifies. After that this call fails instead of reporting a change that cannot take effect, so it cannot be used from a ClientHello good callback.

C.58.4.64 function wolfSSL_enable_cookie

```
int wolfSSL_enable_cookie(
    WOLFSSL * ssl
)
```

Enable server cookies for DTLS 1.2, DTLS 1.3 and TLS 1.3. Like wolfSSL_disable_cookie(), this does not change stateful processing. It can undo a disable before accept begins, including wolfSSL_disable_hrr_cookie(). Missing primary cookie secrets are randomly generated immediately; existing primary and secondary secrets are preserved.

Parameters:

- **ssl** DTLS or TLS 1.3 server session created with wolfSSL_new(). A general-purpose session that has not chosen a side yet is accepted: the policy set here is what the promotion to the server side finds.

See: wolfSSL_disable_cookie

Return:

- WOLFSSL_SUCCESS on success (including an unchanged mode).
- BAD_FUNC_ARG if ssl is NULL or uses an unsupported protocol (TLS 1.2).
- SIDE_ERROR if ssl is a client.
- BAD_STATE_E if the DTLS handshake already committed to stateful processing, as for wolfSSL_disable_cookie(), or if a missing secret must be generated and the session's RNG has been released (as happens when its handshake resources are freed).
- MEMORY_ERROR if secret allocation fails, or another negative error if random secret generation fails. A failure is all or nothing: no secret is installed and the cookie policy is left as it was found, even when the session needs both a DTLS 1.2 and a DTLS 1.3 secret.

C.58.4.65 function wolfSSL_CTX_free

```
void wolfSSL_CTX_free(
    WOLFSSL_CTX * ctx
)
```

This function frees an allocated WOLFSSL_CTX object. This function decrements the CTX reference count and only frees the context when the reference count has reached 0.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().

See:

- wolfSSL_CTX_new
- wolfSSL_new
- wolfSSL_free

Return: none No return.

Example

```
WOLFSSL_CTX* ctx = 0;
...
wolfSSL_CTX_free(ctx);
```

C.58.4.66 function wolfSSL_free

```
void wolfSSL_free(
    WOLFSSL * ssl
)
```

This function frees an allocated wolfSSL object.

Parameters:

- **ssl** pointer to the SSL object, created with wolfSSL_new().

See:

- wolfSSL_CTX_new
- wolfSSL_new
- wolfSSL_CTX_free

Return: none No return.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL* ssl = 0;
...
wolfSSL_free(ssl);
```

C.58.4.67 function wolfSSL_shutdown

```
int wolfSSL_shutdown(
    WOLFSSL * ssl
)
```

This function shuts down an active SSL/TLS connection using the SSL session, ssl. This function will try to send a "close notify" alert to the peer. The calling application can choose to wait for the peer to send its "close notify" alert in response or just go ahead and shut down the underlying connection after directly calling wolfSSL_shutdown (to save resources). Either option is allowed by the TLS specification. If the underlying connection will be used again in the future, the complete two-directional shutdown procedure must be performed to keep synchronization intact between the peers. wolfSSL_shutdown() works with both blocking and non_blocking I/O. When the underlying I/O is non_blocking, wolfSSL_shutdown() will return an error if the underlying I/O could not satisfy the needs of wolfSSL_shutdown() to continue. In this case, a call to wolfSSL_get_error() will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_shutdown() when the underlying I/O is ready.

Parameters:

- **ssl** pointer to the SSL session created with wolfSSL_new().

See:

- wolfSSL_free
- wolfSSL_CTX_free

Return:

- SSL_SUCCESS will be returned upon success.
- SSL_SHUTDOWN_NOT_DONE will be returned when shutdown has not finished, and the function should be called again.
- SSL_FATAL_ERROR will be returned upon failure. Call wolfSSL_get_error() for a more specific error code.

When the connection is already closed or reset and no close notify was ever sent, the exchange can never complete and SSL_FATAL_ERROR is returned, so a loop that calls this function until the shutdown completes terminates. wolfSSL_get_error() then reports SOCKET_PEER_CLOSED_E, unless a more specific error has already been recorded. That code is used whatever closed the connection, including this side sending a fatal alert, so under OPENSSL_EXTRA, where wolfSSL_get_error() reports it as

SSL_ERROR_SYSCALL, a locally aborted connection can surface as a syscall error. Recording it also puts an entry on the OpenSSL error queue in builds that have one, so an application that inspects the queue after tearing down an already_aborted connection finds an entry where it previously found none; call wolfSSL_ERR_clear_error() if leftover entries matter.

Example

```
#include <wolfssl/ssl.h>

int ret = 0;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_shutdown(ssl);
if (ret != 0) {
    // failed to shut down SSL connection
}
```

C.58.4.68 function wolfSSL_SendUserCanceled

```
int wolfSSL_SendUserCanceled(
    WOLFSSL * ssl
)
```

This function sends a user_canceled alert to the peer and then shuts the connection down by calling wolfSSL_shutdown(). It is used when the application abandons a connection for its own reasons rather than because of a protocol failure.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See:

- wolfSSL_shutdown
- wolfSSL_get_error

Return:

- WOLFSSL_SUCCESS on successful shutdown.
- WOLFSSL_SHUTDOWN_NOT_DONE when the peer has yet to send its close notify alert. Call wolfSSL_shutdown() again to complete the bidirectional shutdown. Under WOLFSSL_ERROR_CODE_OPENSSL this value is 0, which is also WOLFSSL_FAILURE, so in that configuration the return value alone does not separate this case from the one below.
- WOLFSSL_FAILURE when ssl is NULL or the alert could not be sent. Call wolfSSL_get_error() for the reason.
- WOLFSSL_FATAL_ERROR when the shutdown that follows the alert fails. Call wolfSSL_get_error() for the reason.
- SSL_SHUTDOWN_ALREADY_DONE_E when the connection was already shut down and WOLFSSL_SHUTDOWNONCE is defined.

Note: RFC 9846, Section 6.1 requires this alert to be followed by a close notify, which is why the shutdown is part of this call. On the receiving side the alert is not itself an error: a TLS 1.3 peer keeps reading until the close notify arrives, whatever AlertLevel was used.

Example

```
int ret = 0;
WOLFSSL* ssl = 0;
...
```

```
ret = wolfSSL_SendUserCanceled(ssl);
if (ret != WOLFSSL_SUCCESS) {
    // failed to shut the connection down, see wolfSSL_get_error()
}
```

C.58.4.69 function wolfSSL_send

```
int wolfSSL_send(
    WOLFSSL * ssl,
    const void * data,
    int sz,
    int flags
)
```

This function writes sz bytes from the buffer, data, to the SSL connection, ssl, using the specified flags for the underlying write operation. If necessary wolfSSL_send() will negotiate an SSL/TLS session if the handshake has not already been performed yet by wolfSSL_connect() or wolfSSL_accept(). wolfSSL_send() works with both blocking and non_blocking I/O. When the underlying I/O is non_blocking, wolfSSL_send() will return when the underlying I/O could not satisfy the needs of wolfSSL_send to continue. In this case, a call to wolfSSL_get_error() will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_send() when the underlying I/O is ready. If the underlying I/O is blocking, wolfSSL_send() will only return once the buffer data of size sz has been completely written or an error occurred.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **data** data buffer to send to peer.
- **sz** size, in bytes, of data to be sent to peer.
- **flags** the send flags to use for the underlying send operation.

See:

- wolfSSL_write
- wolfSSL_read
- wolfSSL_recv

Return:

- 0 the number of bytes written upon success.
- 0 will be returned upon failure. Call wolfSSL_get_error() for the specific error code.
- SSL_FATAL_ERROR will be returned upon failure when either an error occurred or, when using non-blocking sockets, the SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE error was received and and the application needs to call wolfSSL_send() again. Use wolfSSL_get_error() to get a specific error code.
- BAD_FUNC_ARG will be returned when ssl or data is NULL, or sz is negative.

Example

```
WOLFSSL* ssl = 0;
char msg[64] = “hello wolfssl!”;
int msgSz = (int)strlen(msg);
int flags = ... ;
...

input = wolfSSL_send(ssl, msg, msgSz, flags);
if (input != msgSz) {
```

```
    // wolfSSL_send() failed
}
```

C.58.4.70 function wolfSSL_recv

```
int wolfSSL_recv(
    WOLFSSL * ssl,
    void * data,
    int sz,
    int flags
)
```

This function reads sz bytes from the SSL session (ssl) internal read buffer into the buffer data using the specified flags for the underlying recv operation. The bytes read are removed from the internal receive buffer. This function is identical to wolfSSL_read() except that it allows the application to set the recv flags for the underlying read operation. If necessary wolfSSL_recv() will negotiate an SSL/TLS session if the handshake has not already been performed yet by wolfSSL_connect() or wolfSSL_accept(). The SSL/TLS protocol uses SSL records which have a maximum size of 16kB (the max record size can be controlled by the MAX_RECORD_SIZE define in /wolfssl/internal.h). As such, wolfSSL needs to read an entire SSL record internally before it is able to process and decrypt the record. Because of this, a call to wolfSSL_recv() will only be able to return the maximum buffer size which has been decrypted at the time of calling. There may be additional not_yet_decrypted data waiting in the internal wolfSSL receive buffer which will be retrieved and decrypted with the next call to wolfSSL_recv(). If sz is larger than the number of bytes in the internal read buffer, SSL_recv() will return the bytes available in the internal read buffer. If no bytes are buffered in the internal read buffer yet, a call to wolfSSL_recv() will trigger processing of the next record.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **data** buffer where wolfSSL_recv() will place data read.
- **sz** number of bytes to read into data.
- **flags** the recv flags to use for the underlying recv operation.

See:

- wolfSSL_read
- wolfSSL_write
- wolfSSL_peek
- wolfSSL_pending

Return:

- 0 the number of bytes read upon success.
- 0 will be returned upon failure. This may be caused by a either a clean (close notify alert) shutdown or just that the peer closed the connection. Call wolfSSL_get_error() for the specific error code.
- SSL_FATAL_ERROR will be returned upon failure when either an error occurred or, when using non-blocking sockets, the SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE error was received and and the application needs to call wolfSSL_recv() again. Use wolfSSL_get_error() to get a specific error code.
- BAD_FUNC_ARG will be returned when ssl or data is NULL, or sz is negative.

Example

```
WOLFSSL* ssl = 0;
char reply[1024];
```

```
int flags = ... ;
...

input = wolfSSL_recv(ssl, reply, sizeof(reply), flags);
if (input > 0) {
    // “input” number of bytes returned into buffer “reply”
}
```

C.58.4.71 function wolfSSL_get_error

```
int wolfSSL_get_error(
    WOLFSSL * ssl,
    int ret
)
```

This function returns a unique error code describing why the previous API function call (wolfSSL_connect, wolfSSL_accept, wolfSSL_read, wolfSSL_write, etc.) resulted in an error return code (SSL_FAILURE). The return value of the previous function is passed to wolfSSL_get_error through ret. After wolfSSL_get_error is called and returns the unique error code, wolfSSL_ERR_error_string() may be called to get a human_readable error string. See wolfSSL_ERR_error_string() for more information.

Parameters:

- **ssl** pointer to the SSL object, created with wolfSSL_new().
- **ret** return value of the previous function that resulted in an error return code.

See:

- wolfSSL_ERR_error_string
- wolfSSL_ERR_error_string_n
- wolfSSL_ERR_print_errors_fp
- wolfSSL_load_error_strings

Return:

- On successful completion, this function will return the unique error code describing why the previous API function failed.
- SSL_ERROR_NONE will be returned if ret > 0. For ret <= 0, there are some cases when this value can also be returned when a previous API appeared to return an error code but no error actually occurred. An example is calling wolfSSL_read() with a zero sz parameter. A 0 return from wolfSSL_read() usually indicates an error but in this case no error occurred. If wolfSSL_get_error() is called afterwards, SSL_ERROR_NONE will be returned.

Example

```
int err = 0;
WOLFSSL* ssl;
char buffer[80];
...
err = wolfSSL_get_error(ssl, 0);
wolfSSL_ERR_error_string(err, buffer);
printf(“err = %d, %s\n”, err, buffer);
```

C.58.4.72 function wolfSSL_get_alert_history

```
int wolfSSL_get_alert_history(
    WOLFSSL * ssl,
    WOLFSSL_ALERT_HISTORY * h
)
```

This function gets the alert history.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **h** a pointer to a WOLFSSL_ALERT_HISTORY structure that will hold the WOLFSSL struct's alert_history member's value.

See: wolfSSL_get_error

Return: SSL_SUCCESS returned when the function completed successfully. Either there was alert history or there wasn't, either way, the return value is SSL_SUCCESS.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(protocol method);
WOLFSSL* ssl = wolfSSL_new(ctx);
WOLFSSL_ALERT_HISTORY* h;
...
wolfSSL_get_alert_history(ssl, h);
// h now has a copy of the ssl->alert_history  contents
```

C.58.4.73 function wolfSSL_set_session

```
int wolfSSL_set_session(
    WOLFSSL * ssl,
    WOLFSSL_SESSION * session
)
```

This function sets the session to be used when the SSL object, ssl, is used to establish a SSL/TLS connection. For session resumption, before calling wolfSSL_shutdown() with your session object, an application should save the session ID from the object with a call to wolfSSL_get1_session(), which returns a pointer to the session. Later, the application should create a new WOLFSSL object and assign the saved session with wolfSSL_set_session(). At this point, the application may call wolfSSL_connect() and wolfSSL will try to resume the session. The wolfSSL server code allows session resumption by default. The object returned by wolfSSL_get1_session() needs to be freed after the application is done with it by calling wolfSSL_SESSION_free() on it.

Parameters:

- **ssl** pointer to the SSL object, created with wolfSSL_new().
- **session** pointer to the WOLFSSL_SESSION used to set the session for ssl.

See: wolfSSL_get1_session

Return:

- SSL_SUCCESS will be returned upon successfully setting the session.
- SSL_FAILURE will be returned on failure. This could be caused by the session cache being disabled, the session having timed out, or an EMS session being declined because EMS is disabled.
- When OPENSSL_EXTRA and WOLFSSL_ERROR_CODE_OPENSSL are defined, SSL_SUCCESS will be returned even if the session has timed out.

Example

```
int ret;
WOLFSSL* ssl;
WOLFSSL_SESSION* session;
...
session = wolfSSL_get1_session(ssl);
if (session == NULL) {
```

```
    // failed to get session object from ssl object
}
...
ret = wolfSSL_set_session(ssl, session);
if (ret != SSL_SUCCESS) {
    // failed to set the SSL session
}
wolfSSL_SESSION_free(session);
...
```

C.58.4.74 function wolfSSL_get_session

```
WOLFSSL_SESSION * wolfSSL_get_session(
    WOLFSSL * ssl
)
```

Returns a pointer to the current session (WOLFSSL_SESSION) used in ssl. This is a non-persistent pointer to the WOLFSSL_SESSION object, freed when wolfSSL_free is called, so the call should only be used to inspect or modify the current session. For session resumption use wolfSSL_get1_session(), which returns a reference the caller owns and releases with wolfSSL_SESSION_free().

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See:

- wolfSSL_get1_session
- wolfSSL_set_session

Return:

- pointer If successful the call will return a pointer to the the current SSL session object.
- NULL will be returned if ssl is NULL, the SSL session cache is disabled, wolfSSL doesn't have the Session ID available, or mutex functions fail.

This is the default behaviour: settings.h defines NO_SESSION_CACHE_REF unless WOLFSSL_SESSION_CACHE_REF is defined, so every build selects it without asking. The deprecated behaviour remains available by defining WOLFSSL_SESSION_CACHE_REF, or through –enable-session-cache-ref and -DWOLFSSL_SESSION_CACHE_REF=yes, and such a build warns at compile time, both when building the library and at every call to this function; define WOLFSSL_SESSION_CACHE_REF_WARNED to silence both.

Under WOLFSSL_SESSION_CACHE_REF this function instead returns a persistent session object pointer stored in the local cache. The cache size is finite and there is a risk that the session object will be overwritten by another ssl connection by the time the application calls wolfSSL_set_session() on it. Note the lifetime difference when migrating: the deprecated pointer stayed valid after wolfSSL_free(), the default one does not.

Example

```
WOLFSSL* ssl;
WOLFSSL_SESSION* session;
...
session = wolfSSL_get_session(ssl);
if (session == NULL) {
    // failed to get session pointer
}
...
```

C.58.4.75 function wolfSSL_flush_sessions

```
void wolfSSL_flush_sessions(
    WOLFSSL_CTX * ctx,
    long tm
)
```

This function flushes session from the session cache which have expired. The time, tm, is used for the time comparison. Note that wolfSSL currently uses a static table for sessions, so no flushing is needed. As such, this function is currently just a stub. This function provides OpenSSL compatibility (SSL_flush_sessions) when wolfSSL is compiled with the OpenSSL compatibility layer.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **tm** time used in session expiration comparison.

See:

- wolfSSL_get1_session
- wolfSSL_set_session

Return: none No returns.

Example

```
WOLFSSL_CTX* ssl;
...
wolfSSL_flush_sessions(ctx, time(0));
```

C.58.4.76 function wolfSSL_SetServerID

```
int wolfSSL_SetServerID(
    WOLFSSL * ssl,
    const unsigned char * id,
    int len,
    int newSession
)
```

This function associates the client session with the server id. If the newSession flag is on, an existing session won't be reused.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **id** a constant byte pointer that will be copied to the serverID member of the WOLFSSL_SESSION structure.
- **len** an int type representing the length of the session id parameter.
- **newSession** an int type representing the flag to denote whether to reuse a session or not.

See: wolfSSL_set_session

Return:

- SSL_SUCCESS returned if the function executed without error.
- BAD_FUNC_ARG returned if the WOLFSSL struct or id parameter is NULL or if len is not greater than zero.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol );
WOLFSSL* ssl = WOLFSSL_new(ctx);
const byte id[MAX_SIZE];  // or dynamically create space
```

```
int len = 0; // initialize length
int newSession = 0; // flag to allow
…
int ret = wolfSSL_SetServerID(ssl, id, len, newSession);

if (ret == WOLFSSL_SUCCESS) {
    // The Id was successfully set
}
```

C.58.4.77 function wolfSSL_GetSessionIndex

```
int wolfSSL_GetSessionIndex(
    WOLFSSL * ssl
)
```

This function gets the session index of the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_GetSessionAtIndex

Return: int The function returns an int type representing the sessionIndex within the WOLFSSL struct.

Example

```
WOLFSSL_CTX_new( protocol method );
WOLFSSL* ssl = WOLFSSL_new(ctx);
...
int sesIdx = wolfSSL_GetSessionIndex(ssl);

if(sesIdx < 0 || sesIdx > sizeof(ssl->sessionIndex)/sizeof(int)){
    // You have an out of bounds index number and something is not right.
}
```

C.58.4.78 function wolfSSL_GetSessionAtIndex

```
int wolfSSL_GetSessionAtIndex(
    int index,
    WOLFSSL_SESSION * session
)
```

This function gets the session at specified index of the session cache and copies it into memory. The WOLFSSL_SESSION structure holds the session information. The copy is independent of the cache entry: it does not share the ticket buffer, peer certificate or ex_data with the cache, and it stays valid after the cache entry is overwritten or evicted. The caller owns the copy and releases it with wolfSSL_SESSION_free().

Parameters:

- **index** an int type representing the session index.
- **session** a pointer to a WOLFSSL_SESSION structure to copy into, obtained from wolfSSL_SESSION_new().

See:

- UnLockMutex
- LockMutex
- wolfSSL_GetSessionIndex

Return:

- SSL_SUCCESS returned if the function executed successfully and no errors were thrown.
- BAD_MUTEX_E returned if there was an unlock or lock mutex error.
- SSL_FAILURE returned if the function did not execute successfully.

Example

```
int idx; // The index to locate the session.
WOLFSSL_SESSION* session = wolfSSL_SESSION_new();  // Buffer to copy to.
...
if(wolfSSL_GetSessionAtIndex(idx, session) != SSL_SUCCESS){
    // Failure case.
}
...
wolfSSL_SESSION_free(session);
```

C.58.4.79 function wolfSSL_SESSION_get_peer_chain

```
WOLFSSL_X509_CHAIN * wolfSSL_SESSION_get_peer_chain(
    WOLFSSL_SESSION * session
)
```

Returns the peer certificate chain from the WOLFSSL_SESSION struct.

Parameters:

- **session** a pointer to a WOLFSSL_SESSION structure.

See:

- wolfSSL_GetSessionAtIndex
- wolfSSL_GetSessionIndex
- AddSession

Return: pointer A pointer to a WOLFSSL_X509_CHAIN structure that contains the peer certification chain.

Example

```
WOLFSSL_SESSION* session;
WOLFSSL_X509_CHAIN* chain;
...
chain = wolfSSL_SESSION_get_peer_chain(session);
if(!chain){
    // There was no chain. Failure case.
}
```

C.58.4.80 function wolfSSL_CTX_set_verify

```
void wolfSSL_CTX_set_verify(
    WOLFSSL_CTX * ctx,
    int mode,
    VerifyCallback verify_callback
)
```

This function sets the verification method for remote peers and also allows a verify callback to be registered with the SSL context. The verify callback will be called only when a verification failure has occurred. If no verify callback is desired, the NULL pointer can be used for verify_callback. The verification mode of peer certificates is a logically OR'd list of flags. The possible flag values include:

SSL_VERIFY_NONE Client mode: the client will not verify the certificate received from the server and the handshake will continue as normal. Server mode: the server will not send a certificate request to the client. As such, client verification will not be enabled. SSL_VERIFY_PEER Client mode: the client will verify the certificate received from the server during the handshake. This is turned on by default in wolfSSL, therefore, using this option has no effect. Server mode: the server will send a certificate request to the client and verify the client certificate received. SSL_VERIFY_FAIL_IF_NO_PEER_CERT Client mode: no effect when used on the client side. Server mode: the verification will fail on the server side if the client fails to send a certificate when requested to do so (when using SSL_VERIFY_PEER on the SSL server). SSL_VERIFY_FAIL_EXCEPT_PSK Client mode: no effect when used on the client side. Server mode: the verification is the same as SSL_VERIFY_FAIL_IF_NO_PEER_CERT except in the case of a PSK connection. If a PSK connection is being made then the connection will go through without a peer cert. SSL_VERIFY_CLIENT_ONCE Accepted and ignored. It is present so that OpenSSL-derived code compiles unchanged; wolfSSL does not act on it, does not store it, and wolfSSL_get_verify_mode() / wolfSSL_CTX_get_verify_mode() will not report it back. Including it in mode has no effect on any of the other flags.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **mode** flags indicating verification mode for peer's cert.
- **verify_callback** callback to be called when verification fails. If no callback is desired, the NULL pointer can be used for verify_callback.

See: wolfSSL_set_verify

Return: none No return.

Example

```
WOLFSSL_CTX*    ctx    = 0;
...
wolfSSL_CTX_set_verify(ctx, (WOLFSSL_VERIFY_PEER |
                       WOLFSSL_VERIFY_FAIL_IF_NO_PEER_CERT), NULL);
```

C.58.4.81 function wolfSSL_set_verify

```
void wolfSSL_set_verify(
    WOLFSSL * ssl,
    int mode,
    VerifyCallback verify_callback
)
```

This function sets the verification method for remote peers and also allows a verify callback to be registered with the SSL session. The verify callback will be called only when a verification failure has occurred. If no verify callback is desired, the NULL pointer can be used for verify_callback. The verification mode of peer certificates is a logically OR'd list of flags. The possible flag values include: SSL_VERIFY_NONE Client mode: the client will not verify the certificate received from the server and the handshake will continue as normal. Server mode: the server will not send a certificate request to the client. As such, client verification will not be enabled. SSL_VERIFY_PEER Client mode: the client will verify the certificate received from the server during the handshake. This is turned on by default in wolfSSL, therefore, using this option has no effect. Server mode: the server will send a certificate request to the client and verify the client certificate received. SSL_VERIFY_FAIL_IF_NO_PEER_CERT Client mode: no effect when used on the client side. Server mode: the verification will fail on the server side if the client fails to send a certificate when requested to do so (when using SSL_VERIFY_PEER on the SSL server). SSL_VERIFY_FAIL_EXCEPT_PSK Client mode: no effect when used on the client side. Server mode: the verification is the same as SSL_VERIFY_FAIL_IF_NO_PEER_CERT except in the case of a PSK connection. If a PSK connection is being made then the connection will go through without a peer cert. SSL_VERIFY_CLIENT_ONCE Accepted and ignored. It is present so that OpenSSL-derived code

compiles unchanged; wolfSSL does not act on it, does not store it, and wolfSSL_get_verify_mode() / wolfSSL_CTX_get_verify_mode() will not report it back. Including it in mode has no effect on any of the other flags.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **mode** flags indicating verification mode for peer's cert.
- **verify_callback** callback to be called when verification fails. If no callback is desired, the NULL pointer can be used for verify_callback.

See: wolfSSL_CTX_set_verify

Return: none No return.

Example

```
WOLFSSL* ssl = 0;
...
wolfSSL_set_verify(ssl, SSL_VERIFY_PEER | SSL_VERIFY_FAIL_IF_NO_PEER_CERT, 0);
```

C.58.4.82 function wolfSSL_SetCertCbCtx

```
void wolfSSL_SetCertCbCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

This function stores user CTX object information for verify callback.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **ctx** a void pointer that is set to WOLFSSL structure's verifyCbCtx member's value.

See:

- wolfSSL_CTX_save_cert_cache
- wolfSSL_CTX_restore_cert_cache
- wolfSSL_CTX_set_verify

Return: none No return.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
(void*)ctx;
...
if(ssl != NULL){
wolfSSL_SetCertCbCtx(ssl, ctx);
} else {
    // Error case, the SSL is not initialized properly.
}
```

C.58.4.83 function wolfSSL_CTX_SetCertCbCtx

```
void wolfSSL_CTX_SetCertCbCtx(
    WOLFSSL_CTX * ctx,
    void * userCtx
)
```

This function stores user CTX object information for verify callback.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure.
- **userCtx** a void pointer that is used to set WOLFSSL_CTX structure's verifyCbCtx member's value.

See:

- wolfSSL_CTX_save_cert_cache
- wolfSSL_CTX_restore_cert_cache
- wolfSSL_CTX_set_verify

Return: none No return.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
void* userCtx = NULL; // Assign some user defined context
...
if(ctx != NULL){
    wolfSSL_SetCertCbCtx(ctx, userCtx);
} else {
    // Error case, the SSL is not initialized properly.
}
```

C.58.4.84 function wolfSSL_CTX_SetChainVerifyCb

```
int wolfSSL_CTX_SetChainVerifyCb(
    WOLFSSL_CTX * ctx,
    ChainVerifyCb cb
)
```

Replaces wolfSSL's verification of the peer's certificate chain with an application callback, for every SSL/TLS object created from the context. When a callback is set, wolfSSL decodes the certificates from the Certificate message and hands them to the callback as raw DER, the peer's own certificate first. It builds no chain, verifies no signature and checks no date, revocation status, key usage or host name, and the verify callback set with wolfSSL_CTX_set_verify() is not called; the callback is consulted for any certificate the peer sends, even under WOLFSSL_VERIFY_NONE. A server only requests a client certificate under WOLFSSL_VERIFY_PEER and only requires one with WOLFSSL_VERIFY_FAIL_IF_NO_PEER_CERT, so a server that relies on the callback for client authentication must set both. Malformed DER, a certificate the parser refuses regardless of the verify mode such as one with a zero serial number, or one whose decoding stops part way such as one with an unsupported key algorithm, still fails the handshake before the callback is called. An unknown critical extension or a key usage inconsistent with the basic constraints is left for the callback to judge. The callback returns 0 to accept, CHAIN_VERIFY_WANT_E to suspend the handshake until the application re_enters wolfSSL_connect(), wolfSSL_accept(), wolfSSL_read() or wolfSSL_write() (a certificate received after the handshake is resumed by wolfSSL_read() only, and wolfSSL_write() fails with CHAIN_VERIFY_WANT_E until then), or any other value to reject with CHAIN_VERIFY_CB_E and a fatal bad_certificate alert; 0 is the only accepting value, so returning WOLFSSL_SUCCESS rejects. Because wolfSSL checks nothing itself, wolfSSL_get_verify_result() reports WOLFSSL_X509_V_OK once the callback accepts, recording its verdict rather than a verification wolfSSL performed. DTLS, raw public keys and verifying a stapled OCSP response are not supported with the callback: setting it on a context configured for one of them fails, and so does the handshake of a connection using one of them, with CHAIN_VERIFY_UNSUPPORTED_E. The stapling check covers the client, the side that verifies a stapled response; a server may still staple its own status. Requires WOLFSSL_CHAIN_VERIFY_CB (–enable_chain_verify_cb).

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **cb** the callback, or NULL to clear it.

See:

- wolfSSL_SetChainVerifyCb
- wolfSSL_CTX_SetChainVerifyCtx
- wolfSSL_SetChainVerifyCtx
- wolfSSL_GetChainVerifyCtx

Return:

- WOLFSSL_SUCCESS on success.
- BAD_FUNC_ARG when ctx is NULL.
- CHAIN_VERIFY_UNSUPPORTED_E when the context uses a DTLS method, raw public keys or client-side OCSP stapling.

Example

```
static int myChainVerify(WOLFSSL* ssl, const WOLFSSL_BUFFER_INFO* certs,
                         int certsSz, void* ctx)
{
    // certs[0] is the peer's certificate, the rest is the chain it sent
    if (!hsmVerifyStarted(certs, certsSz))
        return CHAIN_VERIFY_WANT_E; // ask again later
    return hsmVerifyPassed() ? 0 : -1;
}
...
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(method);
if (wolfSSL_CTX_SetChainVerifyCb(ctx, myChainVerify) != WOLFSSL_SUCCESS) {
    // context uses DTLS, raw public keys or client-side stapling
}
```

C.58.4.85 function wolfSSL_SetChainVerifyCb

```
int wolfSSL_SetChainVerifyCb(
    WOLFSSL * ssl,
    ChainVerifyCb cb
)
```

Sets the chain verification callback for one SSL/TLS object. It takes precedence over the callback set on the context with wolfSSL_CTX_SetChainVerifyCb(), which documents the callback's contract.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **cb** the callback, or NULL to fall back to the context's.

See:

- wolfSSL_CTX_SetChainVerifyCb
- wolfSSL_SetChainVerifyCtx

Return:

- WOLFSSL_SUCCESS on success.
- BAD_FUNC_ARG when ssl is NULL.
- CHAIN_VERIFY_UNSUPPORTED_E when the object uses DTLS, raw public keys or client-side OCSP stapling.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
if (wolfSSL_SetChainVerifyCb(ssl, myChainVerify) != WOLFSSL_SUCCESS) {
    // object uses DTLS, raw public keys or client-side stapling
}
```

C.58.4.86 function wolfSSL_CTX_SetChainVerifyCtx

```
void wolfSSL_CTX_SetChainVerifyCtx(
    WOLFSSL_CTX * ctx,
    void * userCtx
)
```

Sets the user context passed to the chain verification callback for every SSL/TLS object created from the context.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **userCtx** the value the callback receives as its ctx argument.

See:

- wolfSSL_CTX_SetChainVerifyCb
- wolfSSL_SetChainVerifyCtx
- wolfSSL_GetChainVerifyCtx

Return: none No return.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(method);
wolfSSL_CTX_SetChainVerifyCb(ctx, myChainVerify);
wolfSSL_CTX_SetChainVerifyCtx(ctx, &myHsm);
```

C.58.4.87 function wolfSSL_SetChainVerifyCtx

```
void wolfSSL_SetChainVerifyCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

Sets the user context passed to the chain verification callback for one SSL/TLS object. It takes precedence over the value set on the context with wolfSSL_CTX_SetChainVerifyCtx().

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **ctx** the value the callback receives as its ctx argument, or NULL to fall back to the context's.

See:

- wolfSSL_CTX_SetChainVerifyCtx
- wolfSSL_GetChainVerifyCtx

Return: none No return.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
wolfSSL_SetChainVerifyCtx(ssl, &myConnectionState);
```

C.58.4.88 function wolfSSL_GetChainVerifyCtx

```
void * wolfSSL_GetChainVerifyCtx(
    const WOLFSSL * ssl
)
```

Returns the user context the chain verification callback is called with for this SSL/TLS object: the value set with wolfSSL_SetChainVerifyCtx() when there is one, otherwise the context's.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See:

- wolfSSL_SetChainVerifyCtx
- wolfSSL_CTX_SetChainVerifyCtx

Return: void* the user context, or NULL when none is set or ssl is NULL.

Example

```
struct myState* st = (struct myState*)wolfSSL_GetChainVerifyCtx(ssl);
```

C.58.4.89 function wolfSSL_pending

```
int wolfSSL_pending(
    WOLFSSL * ssl
)
```

This function returns the number of bytes which are buffered and available in the SSL object to be read by wolfSSL_read().

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See:

- wolfSSL_recv
- wolfSSL_read
- wolfSSL_peek

Return:

- int This function returns the number of bytes pending.
- SSL_FAILURE will be returned when ssl is NULL.

Example

```
int pending = 0;
WOLFSSL* ssl = 0;
...

pending = wolfSSL_pending(ssl);
printf("There are %d bytes buffered and available for reading", pending);
```

C.58.4.90 function wolfSSL_load_error_strings

```
void wolfSSL_load_error_strings(
    void
)
```

This function is for OpenSSL compatibility (SSL_load_error_string) only and takes no action.

Parameters:

- **none** No parameters.

See:

- wolfSSL_get_error
- wolfSSL_ERR_error_string
- wolfSSL_ERR_error_string_n
- wolfSSL_ERR_print_errors_fp
- wolfSSL_load_error_strings

Return: none No returns.

Example

```
wolfSSL_load_error_strings();
```

C.58.4.91 function wolfSSL_library_init

```
int wolfSSL_library_init(
    void
)
```

This function is called internally in wolfSSL_CTX_new(). This function is a wrapper around wolfSSL_Init() and exists for OpenSSL compatibility (SSL_library_init) when wolfSSL has been compiled with OpenSSL compatibility layer. wolfSSL_Init() is the more typically-used wolfSSL initialization function.

Parameters:

- **none** No parameters.

See:

- wolfSSL_Init
- wolfSSL_Cleanup

Return:

- SSL_SUCCESS If successful the call will return.
- SSL_FATAL_ERROR is returned upon failure.

Example

```
int ret = 0;
ret = wolfSSL_library_init();
if (ret != SSL_SUCCESS) {
    failed to initialize wolfSSL
}
...
```

C.58.4.92 function wolfSSL_SetDevId

```
int wolfSSL_SetDevId(
    WOLFSSL * ssl,
    int devId
)
```

This function sets the Device Id at the WOLFSSL session level.

Parameters:

- **ssl** pointer to a SSL object, created with wolfSSL_new().
- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID (-2) if not used

See:

- wolfSSL_CTX_SetDevId
- wolfSSL_CTX_GetDevId

Return:

- WOLFSSL_SUCCESS upon success.
- BAD_FUNC_ARG if ssl is NULL.

Example

```
WOLFSSL* ssl;
int DevId = -2;

wolfSSL_SetDevId(ssl, devId);
```

C.58.4.93 function wolfSSL_CTX_SetDevId

```
int wolfSSL_CTX_SetDevId(
    WOLFSSL_CTX * ctx,
    int devId
)
```

This function sets the Device Id at the WOLFSSL_CTX context level.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID (-2) if not used

See:

- wolfSSL_SetDevId
- wolfSSL_CTX_GetDevId

Return:

- WOLFSSL_SUCCESS upon success.
- BAD_FUNC_ARG if ssl is NULL.

Example

```
WOLFSSL_CTX* ctx;
int DevId = -2;

wolfSSL_CTX_SetDevId(ctx, devId);
```

C.58.4.94 function wolfSSL_CTX_GetDevId

```
int wolfSSL_CTX_GetDevId(
    WOLFSSL_CTX * ctx,
    WOLFSSL * ssl
)
```

This function retrieves the Device Id.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().

- **ssl** pointer to a SSL object, created with wolfSSL_new().

See:

- wolfSSL_SetDevId
- wolfSSL_CTX_SetDevId

Return:

- devId upon success.
- INVALID_DEVID if both ssl and ctx are NULL.

Example

```
WOLFSSL_CTX* ctx;

wolfSSL_CTX_GetDevId(ctx, ssl);
```

C.58.4.95 function wolfSSL_CTX_set_session_cache_mode

```
long wolfSSL_CTX_set_session_cache_mode(
    WOLFSSL_CTX * ctx,
    long mode
)
```

This function enables or disables SSL session caching. Behavior depends on the value used for mode. The following values for mode are available: SSL_SESS_CACHE_OFF- disable session caching. Session caching is turned on by default. SSL_SESS_CACHE_NO_AUTO_CLEAR - Disable auto-flushing of the session cache. Auto-flushing is turned on by default.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **mode** modifier used to change behavior of the session cache.

See:

- wolfSSL_flush_sessions
- wolfSSL_get1_session
- wolfSSL_set_session
- wolfSSL_get_sessionID
- wolfSSL_CTX_set_timeout

Return: SSL_SUCCESS will be returned upon success.

Example

```
WOLFSSL_CTX* ctx = 0;
...
ret = wolfSSL_CTX_set_session_cache_mode(ctx, SSL_SESS_CACHE_OFF);
if (ret != SSL_SUCCESS) {
    // failed to turn SSL session caching off
}
```

C.58.4.96 function wolfSSL_CTX_sess_set_get_cb

```
void wolfSSL_CTX_sess_set_get_cb(
    WOLFSSL_CTX * ctx,
    WOLFSSL_SESSION *(*)(WOLFSSL *ssl, const unsigned char *, int, int *) f
)
```

This function registers the callback wolfSSL uses to look a session up in an application managed (external) session cache. It is called on the server side when a peer offers a session id to resume, before the internal cache is consulted. Compatibility layer equivalent of SSL_CTX_sess_set_get_cb().

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **f** the lookup callback, or NULL to remove a previously registered one. It is passed the WOLFSSL object being negotiated, the session id to look up and its length, and a pointer to a copy flag. It returns the matching WOLFSSL_SESSION, or NULL when the cache holds no session for that id. wolfSSL sets the copy flag to 1 before the call. Left at 1, the callback keeps its reference and wolfSSL never releases it, so the session must stay valid while wolfSSL processes the lookup. For TLS 1.3 stateful ticket resumption, which keeps the session longer, wolfSSL takes a reference of its own and releases it when done; a session it cannot take a reference on is then treated as no session found. Set to 0, the callback hands its reference to wolfSSL, which releases it once done with the session.

See:

- wolfSSL_CTX_set_session_cache_mode
- wolfSSL_get1_session
- wolfSSL_SESSION_free
- wolfSSL_flush_sessions

Return: none No return.

Example

```
WOLFSSL_SESSION* myGetSession(WOLFSSL* ssl, const unsigned char* id,
                              int idLen, int* copy)
{
    (void)ssl;
    // The cache keeps its session, so leave copy alone.
    return myCacheLookup(id, idLen);
}
...
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(method);
wolfSSL_CTX_sess_set_get_cb(ctx, myGetSession);
```

C.58.4.97 function wolfSSL_set_session_secret_cb

```
int wolfSSL_set_session_secret_cb(
    WOLFSSL * ssl,
    SessionSecretCb cb,
    void * ctx
)
```

This function sets the session secret callback function. The SessionSecretCb type has the signature: int (*SessionSecretCb)(WOLFSSL* ssl, void* secret, int* secretSz, void* ctx). The sessionSecretCb member of the WOLFSSL struct is set to the parameter cb.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **cb** a SessionSecretCb type that is a function pointer with the above signature.
- **ctx** a pointer to the user context to be stored

See: SessionSecretCb

Return:

- SSL_SUCCESS returned if the execution of the function did not return an error.
- SSL_FATAL_ERROR returned if the WOLFSSL structure is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
// Signature of SessionSecretCb
int SessionSecretCB (WOLFSSL* ssl, void* secret, int* secretSz,
void* ctx) = SessionSecretCb;
…
int wolfSSL_set_session_secret_cb(ssl, SessionSecretCB, (void*)ssl->ctx){
    // Function body.
}
```

C.58.4.98 function wolfSSL_save_session_cache

```
int wolfSSL_save_session_cache(
    const char * fname
)
```

This function persists the session cache to file. It doesn't use memsave because of additional memory use.

Parameters:

- **fname** is a constant char pointer that points to a file for writing.

See:

- XFWRITE
- wolfSSL_restore_session_cache
- wolfSSL_memrestore_session_cache

Return:

- SSL_SUCCESS returned if the function executed without error. The session cache has been written to a file.
- SSL_BAD_FILE returned if fname cannot be opened or is otherwise corrupt.
- FWRITE_ERROR returned if XFWRITE failed to write to the file.
- BAD_MUTEX_E returned if there was a mutex lock failure.

Warning: The file holds session master secrets and resumption credentials in the clear. On POSIX systems it is created with mode 0600; on other platforms the permissions are whatever the port's XFOPEN produces, so the caller must place the file where only the intended user can read it. An existing file is additionally tightened with fchmod, which fails on filesystems without permission bits (FAT); define WOLFSSL_NO_FCHMOD to skip that step on such targets.

Example

```
const char* fname;
...
if(wolfSSL_save_session_cache(fname) != SSL_SUCCESS){
    // Fail to write to file.
}
```

C.58.4.99 function wolfSSL_restore_session_cache

```
int wolfSSL_restore_session_cache(
    const char * fname
)
```

This function restores the persistent session cache from file. It does not use memstore because of additional memory use.

Parameters:

- **fname** a constant char pointer file input that will be read.

See:

- XFREAD
- XFOPEN

Return:

- SSL_SUCCESS returned if the function executed without error.
- SSL_BAD_FILE returned if the file passed into the function was corrupted and could not be opened by XFOPEN.
- FREAD_ERROR returned if the file had a read error from XFREAD.
- CACHE_MATCH_ERROR returned if the session cache header match failed.
- BAD_MUTEX_E returned if there was a mutex lock failure.

Example

```
const char *fname;
...
if(wolfSSL_restore_session_cache(fname) != SSL_SUCCESS){
    // Failure case. The function did not return SSL_SUCCESS.
}
```

C.58.4.100 function wolfSSL_memsave_session_cache

```
int wolfSSL_memsave_session_cache(
    void * mem,
    int sz
)
```

This function persists session cache to memory.

Parameters:

- **mem** a void pointer representing the destination for the memory copy, XMEMCPY().
- **sz** an int type representing the size of mem.

See:

- XMEMCPY
- wolfSSL_get_session_cache_memsize

Return:

- SSL_SUCCESS returned if the function executed without error. The session cache has been successfully persisted to memory.
- BAD_MUTEX_E returned if there was a mutex lock error.
- BUFFER_E returned if the buffer size was too small.

Example

```
void* mem;
int sz; // Max size of the memory buffer.
```

```
…
if(wolfSSL_memsave_session_cache(mem, sz) != SSL_SUCCESS){
    // Failure case, you did not persist the session cache to memory
}
```

C.58.4.101 function wolfSSL_memrestore_session_cache

```
int wolfSSL_memrestore_session_cache(
    const void * mem,
    int sz
)
```

This function restores the persistent session cache from memory.

Parameters:

- **mem** a constant void pointer containing the source of the restoration.
- **sz** an integer representing the size of the memory buffer.

See: wolfSSL_save_session_cache

Return:

- SSL_SUCCESS returned if the function executed without an error.
- BUFFER_E returned if the memory buffer is too small.
- BAD_MUTEX_E returned if the session cache mutex lock failed.
- CACHE_MATCH_ERROR returned if the session cache header match failed.

Example

```
const void* memoryFile;
int szMf;
...
if(wolfSSL_memrestore_session_cache(memoryFile, szMf) != SSL_SUCCESS){
    // Failure case. SSL_SUCCESS was not returned.
}
```

C.58.4.102 function wolfSSL_get_session_cache_memsize

```
int wolfSSL_get_session_cache_memsize(
    void
)
```

This function returns how large the session cache save buffer should be.

Parameters:

- **none** No parameters.

See: wolfSSL_memrestore_session_cache

Return: int This function returns an integer that represents the size of the session cache save buffer.

Example

```
int sz = // Minimum size for error checking;
...
if(sz < wolfSSL_get_session_cache_memsize()){
    // Memory buffer is too small
}
```

C.58.4.103 function wolfSSL_CTX_save_cert_cache

```
int wolfSSL_CTX_save_cert_cache(
    WOLFSSL_CTX * ctx,
    const char * fname
)
```

This function writes the cert cache from memory to file.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, holding the certificate information.
- **fname** a constant char pointer that points to a file for writing.

See:

- CM_SaveCertCache
- DoMemSaveCertCache

Return:

- SSL_SUCCESS if CM_SaveCertCache exits normally.
- BAD_FUNC_ARG is returned if either of the arguments are NULL.
- SSL_BAD_FILE if the cert cache save file could not be opened.
- BAD_MUTEX_E if the lock mutex failed.
- MEMORY_E the allocation of memory failed.
- FWRITE_ERROR Certificate cache file write failed.

Example

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new( protocol def );
const char* fname;
...
if(wolfSSL_CTX_save_cert_cache(ctx, fname)){
    // file was written.
}
```

C.58.4.104 function wolfSSL_CTX_restore_cert_cache

```
int wolfSSL_CTX_restore_cert_cache(
    WOLFSSL_CTX * ctx,
    const char * fname
)
```

This function persistes certificate cache from a file.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, holding the certificate information.
- **fname** a constant char pointer that points to a file for reading.

See:

- CM_RestoreCertCache
- XFOPEN

Return:

- SSL_SUCCESS returned if the function, CM_RestoreCertCache, executes normally.
- SSL_BAD_FILE returned if XFOPEN returns XBADFILE. The file is corrupted.
- MEMORY_E returned if the allocated memory for the temp buffer fails.
- BAD_FUNC_ARG returned if fname or ctx have a NULL value.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
const char* fname = "path to file";
...
if(wolfSSL_CTX_restore_cert_cache(ctx, fname)){
    // check to see if the execution was successful
}
```

C.58.4.105 function wolfSSL_CTX_memsave_cert_cache

```
int wolfSSL_CTX_memsave_cert_cache(
    WOLFSSL_CTX * ctx,
    void * mem,
    int sz,
    int * used
)
```

This function persists the certificate cache to memory.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **mem** a void pointer to the destination (output buffer).
- **sz** the size of the output buffer.
- **used** a pointer to size of the cert cache header.

See:

- DoMemSaveCertCache
- GetCertCacheMemSize
- CM_MemRestoreCertCache
- CM_GetCertCacheMemSize

Return:

- SSL_SUCCESS returned on successful execution of the function. No errors were thrown.
- BAD_MUTEX_E mutex error where the WOLFSSL_CERT_MANAGER member caLock was not 0 (zero).
- BAD_FUNC_ARG returned if ctx, mem, or used is NULL or if sz is less than or equal to 0 (zero).
- BUFFER_E output buffer mem was too small.

Example

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new( protocol );
void* mem;
int sz;
int* used;
...
if(wolfSSL_CTX_memsave_cert_cache(ctx, mem, sz, used) != SSL_SUCCESS){
    // The function returned with an error
}
```

C.58.4.106 function wolfSSL_CTX_memrestore_cert_cache

```
int wolfSSL_CTX_memrestore_cert_cache(
    WOLFSSL_CTX * ctx,
    const void * mem,
```

```
    int sz
)
```

This function restores the certificate cache from memory.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **mem** a void pointer with a value that will be restored to the certificate cache.
- **sz** an int type that represents the size of the mem parameter.

See: CM_MemRestoreCertCache

Return:

- SSL_SUCCESS returned if the function and subroutines executed without an error.
- BAD_FUNC_ARG returned if the ctx or mem parameters are NULL or if the sz parameter is less than or equal to zero.
- BUFFER_E returned if the cert cache memory buffer is too small.
- CACHE_MATCH_ERROR returned if there was a cert cache header mismatch.
- BAD_MUTEX_E returned if the lock mutex on failed.

Example

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new( protocol method );
WOLFSSL* ssl = WOLFSSL_new(ctx);
void* mem;
int sz = (*int) sizeof(mem);
…
if(wolfSSL_CTX_memrestore_cert_cache(ssl->ctx, mem, sz)){
    // The success case
}
```

C.58.4.107 function wolfSSL_CTX_get_cert_cache_memsize

```
int wolfSSL_CTX_get_cert_cache_memsize(
    WOLFSSL_CTX * ctx
)
```

Returns the size the certificate cache save buffer needs to be.

Parameters:

- **ctx** a pointer to a wolfSSL_CTX structure, created using wolfSSL_CTX_new().

See: CM_GetCertCacheMemSize

Return:

- int integer value returned representing the memory size upon success.
- BAD_FUNC_ARG is returned if the WOLFSSL_CTX struct is NULL.
- BAD_MUTEX_E - returned if there was a mutex lock error.

Example

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(protocol);
...
int certCacheSize = wolfSSL_CTX_get_cert_cache_memsize(ctx);

if(certCacheSize != BAD_FUNC_ARG || certCacheSize != BAD_MUTEX_E){
// Successfully retrieved the memory size.
}
```

C.58.4.108 function wolfSSL_CTX_set_cipher_list

```
int wolfSSL_CTX_set_cipher_list(
    WOLFSSL_CTX * ctx,
    const char * list
)
```

This function sets the cipher suite list for a given WOLFSSL_CTX. The list becomes the default cipher suite list for any new WOLFSSL sessions created from the context, and the order in the string is the local preference order from highest to lowest. Each call replaces the previous list. The list is a null-terminated, colon-delimited text string of suite names and/or OpenSSL-style group keywords, for example "TLS13-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:DHE-RSA-AES256-SHA256".

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **list** null-terminated text string and a colon-delimited list of cipher suites and/or keywords to use with the specified SSL context.

See:

- wolfSSL_set_cipher_list
- wolfSSL_get_ciphers
- wolfSSL_get_cipher_list
- wolfSSL_CTX_new

Return:

- WOLFSSL_SUCCESS will be returned upon successful function completion.
- WOLFSSL_FAILURE will be returned on failure.

Each token in the colon-delimited list is one of the following:

1. A specific cipher suite name. wolfSSL accepts both its own short name and the IANA name (when WOLFSSL_NO_ERROR_STRINGS is not defined). For example "ECDHE-RSA-AES128-GCM-SHA256" and "TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256" are equivalent. The complete authoritative list of accepted names lives in the cipher_names[] array in src/internal.c; programs may also enumerate the suites compiled into the current build at runtime via wolfSSL_get_ciphers().
2. An OpenSSL-compatible keyword that selects a family of suites or toggles a class on/off (see the keyword table below). Some keywords require OPENSSL_EXTRA or OPENSSL_ALL to be defined.
3. A negated keyword "!keyword" to disallow a class (requires OPENSSL_EXTRA or OPENSSL_ALL). For example "HIGH:!aNULL".

A "+" operator (e.g. "ECDHE+AESGCM") is recognized only to extract the leading public-key family token ("ECDHE", "RSA" or "DHE"); trailing parts after "+" are ignored by wolfSSL. To intersect with a specific cipher use the explicit suite name instead.

OpenSSL-compatible group keywords:

Keyword	Effect	Required build option
DEFAULT / ALL	Include all built suites; "ALL" also enables anonymous (aNULL)	OPENSSL_EXTRA / OPENSSL_ALL (also accepted as the entire string with no other tokens)
HIGH	All suites except static, anonymous, and NULL ciphers	OPENSSL_EXTRA / OPENSSL_ALL (also accepted as the entire string with no other tokens)

Keyword	Effect	Required build option
LOW / MEDIUM	Accepted but do not restrict by bit size; behave as “RSA”	OPENSSL_EXTRA / OPENSSL_ALL
aNULL	Anonymous (no authentication) suites	OPENSSL_EXTRA / OPENSSL_ALL; suites require HAVE_ANON
eNULL / NULL	Null encryption suites	OPENSSL_EXTRA / OPENSSL_ALL; suites require HAVE_NULL_CIPHER
kDH	Static DH key exchange	OPENSSL_EXTRA / OPENSSL_ALL
DHE / EDH	Ephemeral DH key exchange	OPENSSL_EXTRA / OPENSSL_ALL; suites require !NO_DH
ECDHE / EECDH	Ephemeral ECDH key exchange	OPENSSL_EXTRA / OPENSSL_ALL; suites require HAVE_ECC
kRSA / RSA	Static RSA key exchange	OPENSSL_EXTRA / OPENSSL_ALL; suites require !NO_RSA
PSK	Pre-shared-key suites	OPENSSL_EXTRA / OPENSSL_ALL; suites require !NO_PSK
DSS	Silently ignored — wolfSSL has no DSA ciphersuites	OPENSSL_EXTRA / OPENSSL_ALL
EXP / EXPORT	Silently ignored — export-grade ciphers are not supported	OPENSSL_EXTRA / OPENSSL_ALL
AES128 / SHA1 / RC4	When negated (“!AES128”, etc.), disable that class	WOLFSSL_SYS_CRYPTO_POLICY
@SECLEVEL=n	Set OpenSSL-compatible security level	WOLFSSL_SYS_CRYPTO_POLICY

Representative TLS 1.3 cipher suite names (each guarded by its own BUILD_* macro; most are enabled automatically by –enable-tls13):

Name (wolfSSL)	IANA name	Required build option
TLS13-AES128-GCM-SHA256	TLS_AES_128_GCM_SHA256	BUILD_TLS_AES_128_GCM_SHA256 (default with TLS 1.3)
TLS13-AES256-GCM-SHA384	TLS_AES_256_GCM_SHA384	BUILD_TLS_AES_256_GCM_SHA384
TLS13-CHACHA20-POLY1305-SHA256	TLS_CHACHA20_POLY1305_SHA256	BUILD_TLS_CHACHA20_POLY1305_SHA256 (HAVE_CHACHA + HAVE_POLY1305)
TLS13-AES128-CCM-SHA256	TLS_AES_128_CCM_SHA256	BUILD_TLS_AES_128_CCM_SHA256 (HAVE_AESCCM)
TLS13-AES128-CCM-8-SHA256	TLS_AES_128_CCM_8_SHA256	BUILD_TLS_AES_128_CCM_8_SHA256 (HAVE_AESCCM)
TLS13-SM4-GCM-SM3	TLS_SM4_GCM_SM3	BUILD_TLS_SM4_GCM_SM3 (WOLFSSL_SM4_GCM + WOLFSSL_SM3)

COPYRIGHT ©2024 wolfSSL Inc.

Name (wolfSSL)	IANA name	Required build option
TLS13-SM4-CCM-SM3	TLS_SM4_CCM_SM3	BUILD_TLS_SM4_CCM_SM3 (WOLFSSL_SM4_CCM + WOLFSSL_SM3)
TLS13-SHA256-SHA256	TLS_SHA256_SHA256	BUILD_TLS_SHA256_SHA256 (integrity-only)
TLS13-SHA384-SHA384	TLS_SHA384_SHA384	BUILD_TLS_SHA384_SHA384

Representative TLS 1.2 cipher suite name families (each individual suite is guarded by its own BUILD_* macro derived from the IANA name; the common build-option requirements are summarized below):

Name family / example	Typical requirements
ECDHE-ECDSA-AES128-GCM-SHA256, ECDHE-ECDSA-AES256-GCM-SHA384, ECDHE-ECDSA-CHACHA20-POLY1305	HAVE_ECC, HAVE_AESGCM (or HAVE_CHACHA + HAVE_POLY1305)
ECDHE-RSA-AES128-GCM-SHA256, ECDHE-RSA-AES256-GCM-SHA384, ECDHE-RSA-CHACHA20-POLY1305	HAVE_ECC, !NO_RSA, HAVE_AESGCM (or HAVE_CHACHA + HAVE_POLY1305)
DHE-RSA-AES128-GCM-SHA256, DHE-RSA-AES256-GCM-SHA384, DHE-RSA-CHACHA20-POLY1305	!NO_DH, !NO_RSA, HAVE_AESGCM (or HAVE_CHACHA + HAVE_POLY1305)
AES128-SHA, AES256-SHA, AES128-SHA256, AES256-SHA256, AES128-GCM-SHA256, AES256-GCM-SHA384 (static-RSA)	!NO_RSA, !NO_AES_CBC and/or HAVE_AESGCM
DES-CBC3-SHA, RC4-SHA, RC4-MD5	Legacy: !NO_DES3 / !NO_RC4, !NO_OLD_TLS
NULL-SHA, NULL-SHA256, NULL-MD5	HAVE_NULL_CIPHER
PSK-AES128-CBC-SHA256, PSK-AES256-GCM-SHA384, ECDHE-PSK-AES128-CBC-SHA256, DHE-PSK-AES256-GCM-SHA384	!NO_PSK (HAVE_ECC for ECDHE-PSK, !NO_DH for DHE-PSK)
ADH-AES128-SHA, ADH-AES256-SHA	HAVE_ANON
ECDHE-ECDSA-SM4-GCM-SM3, ECDHE-ECDSA-SM4-CCM-SM3	WOLFSSL_SM2, WOLFSSL_SM3, WOLFSSL_SM4_GCM/WOLFSSL_SM4_CCM

Notes:

- TLS 1.3 suite names and TLS 1.2 suite names may be mixed in the same list; wolfSSL groups them by version internally.
- For DTLS, RC4-based stream ciphers in the list are silently dropped.
- When set as the literal whole-string “DEFAULT”, “ALL”, “HIGH”, or the empty string, wolfSSL installs its built-in default suite list and returns success without parsing further tokens.

Example

```
WOLFSSL_CTX* ctx = 0;
...
ret = wolfSSL_CTX_set_cipher_list(ctx,
    "TLS13-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:"
    "DHE-RSA-AES256-SHA256");
if (ret != WOLFSSL_SUCCESS) {
    // failed to set cipher suite list
}
```

C.58.4.109 function wolfSSL_set_cipher_list

```
int wolfSSL_set_cipher_list(
    WOLFSSL * ssl,
    const char * list
)
```

This function sets the cipher suite list for a given WOLFSSL session. The list format and the set of recognized suite names and keywords are identical to those documented for wolfSSL_CTX_set_cipher_list(); refer to that function for the full keyword/suite tables and required build options. Each call replaces the previous list on the session.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **list** null-terminated text string and a colon-delimited list of cipher suites and/or keywords to use with the specified SSL session.

See:

- wolfSSL_CTX_set_cipher_list
- wolfSSL_get_ciphers
- wolfSSL_get_cipher_list
- wolfSSL_new

Return:

- WOLFSSL_SUCCESS will be returned upon successful function completion.
- WOLFSSL_FAILURE will be returned on failure.

Example

```
int ret = 0;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_set_cipher_list(ssl,
    "TLS13-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:"
    "DHE-RSA-AES256-SHA256");
if (ret != WOLFSSL_SUCCESS) {
    // failed to set cipher suite list
}
```

C.58.4.110 function wolfSSL_dtls_set_using_nonblock

```
void wolfSSL_dtls_set_using_nonblock(
    WOLFSSL * ssl,
    int nonblock
)
```

This function informs the WOLFSSL DTLS object that the underlying UDP I/O is non-blocking. After an application creates a WOLFSSL object, if it will be used with a non-blocking UDP socket, call wolfSSL_dtls_set_using_nonblock() on it. This lets the WOLFSSL object know that receiving EWOULDBLOCK means that the recvfrom call would block rather than that it timed out.

Parameters:

- **ssl** pointer to the DTLS session, created with wolfSSL_new().
- **nonblock** value used to set non-blocking flag on WOLFSSL object. Use 1 to specify non-blocking, otherwise 0.

See:

- wolfSSL_dtls_get_using_nonblock
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls_get_current_timeout

Return: none No return.

Example

```
WOLFSSL* ssl = 0;
...
wolfSSL_dtls_set_using_nonblock(ssl, 1);
```

C.58.4.111 function wolfSSL_dtls_get_using_nonblock

```
int wolfSSL_dtls_get_using_nonblock(
    WOLFSSL * ssl
)
```

This function allows the application to determine if wolfSSL is using non-blocking I/O with UDP. If wolfSSL is using non-blocking I/O, this function will return 1, otherwise 0. After an application creates a WOLFSSL object, if it will be used with a non-blocking UDP socket, call wolfSSL_dtls_set_using_nonblock() on it. This lets the WOLFSSL object know that receiving EWOULDBLOCK means that the recvfrom call would block rather than that it timed out. This function is only meaningful to DTLS sessions.

Parameters:

- **ssl** pointer to the DTLS session, created with wolfSSL_new().

See:

- wolfSSL_dtls_set_using_nonblock
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls_set_using_nonblock

Return:

- 0 underlying I/O is blocking.
- 1 underlying I/O is non-blocking.

Example

```
int ret = 0;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_dtls_get_using_nonblock(ssl);
if (ret == 1) {
    // underlying I/O is non-blocking
}
...
```

C.58.4.112 function wolfSSL_dtls_get_current_timeout

```
int wolfSSL_dtls_get_current_timeout(
    WOLFSSL * ssl
)
```

This function returns the current timeout value in seconds for the WOLFSSL object. When using non-blocking sockets, something in the user code needs to decide when to check for available recv data and how long it has been waiting. The value returned by this function indicates how long the application should wait.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_dtls
- wolfSSL_dtls_get_peer
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls_set_peer

Return:

- seconds The current DTLS timeout value in seconds
- NOT_COMPILED_IN if wolfSSL was not built with DTLS support.

Example

```
int timeout = 0;
WOLFSSL* ssl;
...
timeout = wolfSSL_get_dtls_current_timeout(ssl);
printf("DTLS timeout (sec) = %d\n", timeout);
```

C.58.4.113 function wolfSSL_dtls13_use_quick_timeout

```
int wolfSSL_dtls13_use_quick_timeout(
    WOLFSSL * ssl
)
```

This function returns true if the application should setup a quicker timeout. When using non-blocking sockets, something in the user code needs to decide when to check for available data and how long it needs to wait. If this function returns true, it means that the library already detected some disruption in the communication, but it wants to wait for a little longer in case some messages from the other peers are still in flight. Is up to the application to fine tune the value of this timer, a good one may be dtls_get_current_timeout() / 4.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_dtls
- wolfSSL_dtls_get_peer
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls_set_peer
- wolfSSL_dtls13_set_send_more_acks

Return: true if the application code should setup a quicker timeout

C.58.4.114 function wolfSSL_dtls13_set_send_more_acks

```
void wolfSSL_dtls13_set_send_more_acks(
    WOLFSSL * ssl,
    int value
)
```

This function sets whether the library should send ACKs to the other peer immediately when detecting disruption or not. Sending ACKs immediately assures minimum latency but it may consume more bandwidth than necessary. If the application manages the timer by itself and this option is set to 0

then application code can use wolfSSL_dtls13_use_quick_timeout() to determine if it should setup a quicker timeout to send those delayed ACKs.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **value** 1 to set the option, 0 to disable the option

See:

- wolfSSL_dtls
- wolfSSL_dtls_get_peer
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls_set_peer
- wolfSSL_dtls13_use_quick_timeout

C.58.4.115 function wolfSSL_dtls_set_timeout_init

```
int wolfSSL_dtls_set_timeout_init(
    WOLFSSL * ssl,
    int timeout
)
```

This function sets the dtls timeout.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **timeout** an int type that will be set to the dtls_timeout_init member of the WOLFSSL structure.

See:

- wolfSSL_dtls_set_timeout_max
- wolfSSL_dtls_got_timeout

Return:

- SSL_SUCCESS returned if the function executes without an error. The dtls_timeout_init and the dtls_timeout members of SSL have been set.
- BAD_FUNC_ARG returned if the WOLFSSL struct is NULL or if the timeout is not greater than 0. It will also return if the timeout argument exceeds the maximum value allowed.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
int timeout = TIMEOUT;
...
if(wolfSSL_dtls_set_timeout_init(ssl, timeout)){
    // the dtls timeout was set
} else {
    // Failed to set DTLS timeout.
}
```

C.58.4.116 function wolfSSL_dtls_set_timeout_max

```
int wolfSSL_dtls_set_timeout_max(
    WOLFSSL * ssl,
    int timeout
)
```

This function sets the maximum dtls timeout.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **timeout** an int type representing the dtls maximum timeout.

See:

- wolfSSL_dtls_set_timeout_init
- wolfSSL_dtls_got_timeout

Return:

- SSL_SUCCESS returned if the function executed without an error.
- BAD_FUNC_ARG returned if the WOLFSSL struct is NULL or if the timeout argument is not greater than zero or is less than the dtls_timeout_init member of the WOLFSSL structure.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
int timeout = TIMEOUTVAL;
...
int ret = wolfSSL_dtls_set_timeout_max(ssl);
if(!ret){
    // Failed to set the max timeout
}
```

C.58.4.117 function wolfSSL_dtls_got_timeout

```
int wolfSSL_dtls_got_timeout(
    WOLFSSL * ssl
)
```

When using non-blocking sockets with DTLS, this function should be called on the WOLFSSL object when the controlling code thinks the transmission has timed out. It performs the actions needed to retry the last transmit, including adjusting the timeout value. If it has been too long, this will return a failure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_dtls_get_current_timeout
- wolfSSL_dtls_get_peer
- wolfSSL_dtls_set_peer
- wolfSSL_dtls

Return:

- SSL_SUCCESS will be returned upon success
- SSL_FATAL_ERROR will be returned if there have been too many retransmissions/timeouts without getting a response from the peer.
- NOT_COMPILED_IN will be returned if wolfSSL was not compiled with DTLS support.

Example

```
See the following files for usage examples:
<wolfssl_root>/examples/client/client.c
<wolfssl_root>/examples/server/server.c
```

C.58.4.118 function wolfSSL_dtls_retransmit

```
int wolfSSL_dtls_retransmit(
    WOLFSSL * ssl
)
```

When using non-blocking sockets with DTLS, this function retransmits the last handshake flight ignoring the expected timeout value and retransmit count. It is useful for applications that are using DTLS and need to manage even the timeout and retry count.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_dtls_get_current_timeout
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls

Return:

- SSL_SUCCESS will be returned upon success
- SSL_FATAL_ERROR will be returned if there have been too many retransmissions/timeouts without getting a response from the peer.

Example

```
int ret = 0;
WOLFSSL* ssl;
...
ret = wolfSSL_dtls_retransmit(ssl);
```

C.58.4.119 function wolfSSL_dtls13_do_scheduled_work

```
int wolfSSL_dtls13_do_scheduled_work(
    WOLFSSL * ssl
)
```

Sends the DTLS 1.3 work that was scheduled while reading. With WOLFSSL_RW_THREADED the read path never transmits, because that would race the write thread over the output buffer and the sending key schedule, so ACKs, retransmissions and a KeyUpdate the peer asked for are only sent from the write side. An application that reads without writing must call this, from the same thread it uses for writing, or those messages are never sent and the peer keeps retransmitting what it is waiting to have acknowledged. Not for write-dup applications, which drain through wolfSSL_write().

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_dtls13_pending_work
- wolfSSL_dtls_retransmit

Return:

- WOLFSSL_SUCCESS on success, including when there was nothing to do.
- WOLFSSL_FATAL_ERROR if ssl is NULL, is not a DTLS 1.3 object, is part of a write-dup pair, or the send failed. Call wolfSSL_get_error() to tell a retryable SSL_ERROR_WANT_WRITE from a real failure.

A KeyUpdate is only sent while none of ours is still unacknowledged, since DTLS must not have two in flight. Completing one we started needs the peer's acknowledgement processed, which rotates the sending keys and so does not happen here, so in a WOLFSSL_RW_THREADED build a peer request arriving after that point is held rather than answered.

A send that could only write part of a record returns WOLFSSL_FATAL_ERROR with wolfSSL_get_error() reporting SSL_ERROR_WANT_WRITE. That is not a failure of the connection: the record is held and the next call sends the rest, so wolfSSL_dtls13_pending_work() keeps reporting work until it is out. Treat it as a retry rather than as a reason to stop draining.

Example

```
WOLFSSL* ssl;
...
while (wolfSSL_dtls13_pending_work(ssl)) {
    if (wolfSSL_dtls13_do_scheduled_work(ssl) != WOLFSSL_SUCCESS) {
        if (wolfSSL_get_error(ssl, 0) == SSL_ERROR_WANT_WRITE) {
            // the socket is full, wait for it and call again
            break;
        }
        // a real error
        break;
    }
}
```

C.58.4.120 function wolfSSL_dtls13_pending_work

```
int wolfSSL_dtls13_pending_work(
    WOLFSSL * ssl
)
```

Reports whether the object has DTLS 1.3 work waiting to be sent by wolfSSL_dtls13_do_scheduled_work(). Only meaningful with WOLFSSL_RW_THREADED. The answer is advisory and can change as soon as it is returned. Only work the pump can actually carry out is reported, so a drain loop over the pair terminates; waiting for the peer to acknowledge a key update we sent is not reported, as there is nothing to send for it. A record the pump could only write in part is reported, so that the retry it owes is not lost.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_dtls13_do_scheduled_work

Return:

- 1 if there is work to send, or if it could not be determined.
- 0 if there is nothing to do, or ssl is not supported here.

Example

```
WOLFSSL* ssl;
...
if (wolfSSL_dtls13_pending_work(ssl))
    wolfSSL_dtls13_do_scheduled_work(ssl);
```

C.58.4.121 function wolfSSL_dtls

```
int wolfSSL_dtls(
    WOLFSSL * ssl
)
```

This function is used to determine if the SSL session has been configured to use DTLS.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_dtls_get_current_timeout
- wolfSSL_dtls_get_peer
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls_set_peer

Return:

- 1 If the SSL session (ssl) has been configured to use DTLS, this function will return 1.
- 0 otherwise.

Example

```
int ret = 0;
WOLFSSL* ssl;
...
ret = wolfSSL_dtls(ssl);
if (ret) {
    // SSL session has been configured to use DTLS
}
```

C.58.4.122 function wolfSSL_dtls_set_peer

```
int wolfSSL_dtls_set_peer(
    WOLFSSL * ssl,
    void * peer,
    unsigned int peerSz
)
```

This function sets the DTLS peer, peer (sockaddr_in) with size of peerSz.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **peer** pointer to peer's sockaddr_in structure. If NULL then the peer information in ssl is cleared.
- **peerSz** size of the sockaddr_in structure pointed to by peer. If 0 then the peer information in ssl is cleared.

See:

- wolfSSL_dtls_get_current_timeout
- wolfSSL_dtls_set_pending_peer
- wolfSSL_dtls_get_peer
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls

Return:

- SSL_SUCCESS will be returned upon success.
- SSL_FAILURE will be returned upon failure.
- SSL_NOT_IMPLEMENTED will be returned if wolfSSL was not compiled with DTLS support.

Example

```
int ret = 0;
WOLFSSL* ssl;
sockaddr_in addr;
...
ret = wolfSSL_dtls_set_peer(ssl, &addr, sizeof(addr));
if (ret != SSL_SUCCESS) {
    // failed to set DTLS peer
}
```

C.58.4.123 function wolfSSL_dtls_set_pending_peer

```
int wolfSSL_dtls_set_pending_peer(
    WOLFSSL * ssl,
    void * peer,
    unsigned int peerSz
)
```

This function sets the pending DTLS peer, peer (sockaddr_in) with size of peerSz. This sets the pending peer that will be upgraded to a regular peer when we successfully de-protect the next record. This is useful in scenarios where the peer's address can change to avoid off-path attackers from changing the peer address. This should be used with Connection ID's to allow seamless and safe transition to a new peer address.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **peer** pointer to peer's sockaddr_in structure. If NULL then the peer information in ssl is cleared.
- **peerSz** size of the sockaddr_in structure pointed to by peer. If 0 then the peer information in ssl is cleared.

See:

- wolfSSL_dtls_get_current_timeout
- wolfSSL_dtls_set_peer
- wolfSSL_dtls_get_peer
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls

Return:

- SSL_SUCCESS will be returned upon success.
- SSL_FAILURE will be returned upon failure.
- SSL_NOT_IMPLEMENTED will be returned if wolfSSL was not compiled with DTLS support.

Example

```
int ret = 0;
WOLFSSL* ssl;
sockaddr_in addr;
...
ret = wolfSSL_dtls_set_pending_peer(ssl, &addr, sizeof(addr));
if (ret != SSL_SUCCESS) {
    // failed to set DTLS peer
}
```

C.58.4.124 function wolfSSL_dtls_get_peer

```
int wolfSSL_dtls_get_peer(
    WOLFSSL * ssl,
    void * peer,
    unsigned int * peerSz
)
```

This function gets the sockaddr_in (of size peerSz) of the current DTLS peer. The function will compare peerSz to the actual DTLS peer size stored in the SSL session. If the peer will fit into peer, the peer's sockaddr_in will be copied into peer, with peerSz set to the size of peer.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **peer** pointer to memory location to store peer's sockaddr_in structure.
- **peerSz** input/output size. As input, the size of the allocated memory pointed to by peer. As output, the size of the actual sockaddr_in structure pointed to by peer.

See:

- wolfSSL_dtls_get_current_timeout
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls_set_peer
- wolfSSL_dtls

Return:

- SSL_SUCCESS will be returned upon success.
- SSL_FAILURE will be returned upon failure.
- SSL_NOT_IMPLEMENTED will be returned if wolfSSL was not compiled with DTLS support.

Example

```
int ret = 0;
WOLFSSL* ssl;
sockaddr_in addr;
...
ret = wolfSSL_dtls_get_peer(ssl, &addr, sizeof(addr));
if (ret != SSL_SUCCESS) {
    // failed to get DTLS peer
}
```

C.58.4.125 function wolfSSL_dtls_get0_peer

```
int wolfSSL_dtls_get0_peer(
    WOLFSSL * ssl,
    const void ** peer,
    unsigned int * peerSz
)
```

This function gets the sockaddr_in (of size peerSz) of the current DTLS peer. This is a zero-copy alternative to wolfSSL_dtls_get_peer().

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **peer** pointer to return the internal buffer holding the peer address
- **peerSz** output the size of the actual sockaddr_in structure pointed to by peer.

See:

- wolfSSL_dtls_get_current_timeout

- wolfSSL_dtls_got_timeout
- wolfSSL_dtls_set_peer
- wolfSSL_dtls

Return:

- SSL_SUCCESS will be returned upon success.
- SSL_FAILURE will be returned upon failure.
- SSL_NOT_IMPLEMENTED will be returned if wolfSSL was not compiled with DTLS support.

Example

```
int ret = 0;
WOLFSSL* ssl;
sockaddr_in* addr;
unsigned int addrSz;
...
ret = wolfSSL_dtls_get_peer(ssl, &addr, &addrSz);
if (ret != SSL_SUCCESS) {
    // failed to get DTLS peer
}
```

C.58.4.126 function wolfSSL_ERR_error_string

```
char * wolfSSL_ERR_error_string(
    unsigned long errNumber,
    char * data
)
```

This function converts an error code returned by wolfSSL_get_error() into a more human_readable error string. errNumber is the error code returned by wolfSSL_get_error() and data is the storage buffer which the error string will be placed in. The maximum length of data is 80 characters by default, as defined by MAX_ERROR_SZ is wolfssl/wolfcrypt/error.h.

Parameters:

- **errNumber** error code returned by wolfSSL_get_error().
- **data** output buffer containing human-readable error string matching errNumber.

See:

- wolfSSL_get_error
- wolfSSL_ERR_error_string_n
- wolfSSL_ERR_print_errors_fp
- wolfSSL_load_error_strings

Return:

- success On successful completion, this function returns the same string as is returned in data.
- failure Upon failure, this function returns a string with the appropriate failure reason, msg.

Example

```
int err = 0;
WOLFSSL* ssl;
char buffer[80];
...
err = wolfSSL_get_error(ssl, 0);
wolfSSL_ERR_error_string(err, buffer);
printf("err = %d, %s\n", err, buffer);
```

C.58.4.127 function wolfSSL_ERR_error_string_n

```
void wolfSSL_ERR_error_string_n(
    unsigned long e,
    char * buf,
    unsigned long len
)
```

This function is a version of wolfSSL_ERR_error_string() where len specifies the maximum number of characters that may be written to buf. Like wolfSSL_ERR_error_string(), this function converts an error code returned from wolfSSL_get_error() into a more human-readable error string. The human-readable string is placed in buf.

Parameters:

- **e** error code returned by wolfSSL_get_error().
- **buff** output buffer containing human-readable error string matching e.
- **len** maximum length in characters which may be written to buf.

See:

- wolfSSL_get_error
- wolfSSL_ERR_error_string
- wolfSSL_ERR_print_errors_fp
- wolfSSL_load_error_strings

Return: none No returns.

Example

```
int err = 0;
WOLFSSL* ssl;
char buffer[80];
...
err = wolfSSL_get_error(ssl, 0);
wolfSSL_ERR_error_string_n(err, buffer, 80);
printf("err = %d, %s\n", err, buffer);
```

C.58.4.128 function wolfSSL_get_shutdown

```
int wolfSSL_get_shutdown(
    const WOLFSSL * ssl
)
```

This function checks the shutdown conditions in closeNotify or connReset or sentNotify members of the Options structure. The Options structure is within the WOLFSSL structure.

Parameters:

- **ssl** a constant pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_SESSION_free

Return:

- 1 SSL_SENT_SHUTDOWN is returned.
- 2 SSL_RECEIVED_SHUTDOWN is returned.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = WOLFSSL_CTX_new( protocol method );
```

```
WOLFSSL* ssl = WOLFSSL_new(ctx);
…
int ret;
ret = wolfSSL_get_shutdown(ssl);

if(ret == 1){
    SSL_SENT_SHUTDOWN
} else if(ret == 2){
    SSL_RECEIVED_SHUTDOWN
} else {
    Fatal error.
}
```

C.58.4.129 function wolfSSL_session_reused

```
int wolfSSL_session_reused(
    WOLFSSL * ssl
)
```

This function returns the resuming member of the options struct. The flag indicates whether or not to reuse a session. If not, a new session must be established.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_SESSION_free
- wolfSSL_GetSessionIndex
- wolfSSL_memsave_session_cache

Return: This function returns an int type held in the Options structure representing the flag for session reuse.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
…
if(!wolfSSL_session_reused(sslResume)){
    // No session reuse allowed.
}
```

C.58.4.130 function wolfSSL_is_init_finished

```
int wolfSSL_is_init_finished(
    const WOLFSSL * ssl
)
```

This function checks to see if the connection is established.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_set_accept_state
- wolfSSL_get_keys
- wolfSSL_set_shutdown

Return:

- 0 returned if the connection is not established, i.e. the WOLFSSL struct is NULL or the handshake is not done.
- 1 returned if the connection is established i.e. the WOLFSSL handshake is done.

EXAMPLE

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_is_init_finished(ssl)){
    Handshake is done and connection is established
}
```

C.58.4.131 function wolfSSL_get_version

```
const char * wolfSSL_get_version(
    WOLFSSL * ssl
)
```

Returns the SSL version being used as a string.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_lib_version

Return:

- “SSLv3” Using SSLv3
- “TLSv1” Using TLSv1
- “TLSv1.1” Using TLSv1.1
- “TLSv1.2” Using TLSv1.2
- “TLSv1.3” Using TLSv1.3
- “DTLS”: Using DTLS
- “DTLSv1.2” Using DTLSv1.2
- “unknown” There was a problem determining which version of TLS being used.

Example

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
WOLFSSL_METHOD method = // Some wolfSSL method
ctx = wolfSSL_CTX_new(method);
ssl = wolfSSL_new(ctx);
printf(wolfSSL_get_version("Using version: %s", ssl));
```

C.58.4.132 function wolfSSL_get_current_cipher_suite

```
int wolfSSL_get_current_cipher_suite(
    WOLFSSL * ssl
)
```

Returns the current cipher suit an ssl session is using.

Parameters:

- **ssl** The SSL session to check.

See:

- wolfSSL_CIPHER_get_name
- wolfSSL_get_current_cipher
- wolfSSL_get_cipher_list

Return:

- ssl->options.cipherSuite An integer representing the current cipher suite.
- 0 The ssl session provided is null.

Example

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
WOLFSSL_METHOD method = // Some wolfSSL method
ctx = wolfSSL_CTX_new(method);
ssl = wolfSSL_new(ctx);

if(wolfSSL_get_current_cipher_suite(ssl) == 0)
{
    // Error getting cipher suite
}
```

C.58.4.133 function wolfSSL_get_current_cipher

```
WOLFSSL_CIPHER * wolfSSL_get_current_cipher(
    WOLFSSL * ssl
)
```

This function returns a pointer to the current cipher in the ssl session. The cipher is filled in from the session on each call, so call it again after the session changes, e.g. after the handshake.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_get_cipher
- wolfSSL_get_cipher_name_internal
- wolfSSL_get_cipher_name

Return:

- The function returns the address of the cipher member of the WOLFSSL struct. This is a pointer to the WOLFSSL_CIPHER structure.
- NULL returned if the WOLFSSL structure is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
…
WOLFSSL_CIPHER* cipherCurr = wolfSSL_get_current_cipher;

if(!cipherCurr){
    // Failure case.
} else {
```

```
    // The cipher was returned to cipherCurr
}
```

C.58.4.134 function wolfSSL_CIPHER_get_name

```
const char * wolfSSL_CIPHER_get_name(
    const WOLFSSL_CIPHER * cipher
)
```

This function matches the cipher suite in the SSL object with the available suites and returns the string representation.

Parameters:

- **cipher** a constant pointer to a WOLFSSL_CIPHER structure.

See:

- wolfSSL_get_cipher
- wolfSSL_get_current_cipher
- wolfSSL_get_cipher_name_internal
- wolfSSL_get_cipher_name

Return:

- string This function returns the string representation of the matched cipher suite.
- none It will return “None” if there are no suites matched.

Example

```
// gets cipher name in the format DHE_RSA ...
const char* wolfSSL_get_cipher_name_internal(WOLFSSL* ssl){
WOLFSSL_CIPHER* cipher;
const char* fullName;
…
cipher = wolfSSL_get_curent_cipher(ssl);
fullName = wolfSSL_CIPHER_get_name(cipher);

if(fullName){
    // sanity check on returned cipher
}
```

C.58.4.135 function wolfSSL_get_cipher

```
const char * wolfSSL_get_cipher(
    WOLFSSL *
)
```

This function matches the cipher suite in the SSL object with the available suites.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_CIPHER_get_name
- wolfSSL_get_current_cipher

Return: This function returns the string value of the suite matched. It will return “None” if there are no suites matched.

Example

```
#ifdef WOLFSSL_DTLS
…
// make sure a valid suite is used
if(wolfSSL_get_cipher(ssl) == NULL){
    WOLFSSL_MSG("Can not match cipher suite imported");
    return MATCH_SUITE_ERROR;
}
…
#endif // WOLFSSL_DTLS
```

C.58.4.136 function wolfSSL_get1_session

```
WOLFSSL_SESSION * wolfSSL_get1_session(
    WOLFSSL * ssl
)
```

This function returns the WOLFSSL_SESSION from the WOLFSSL structure as a reference type. This requires calling wolfSSL_SESSION_free to release the session reference. The WOLFSSL_SESSION pointed to contains all the necessary information required to perform a session resumption and reestablish the connection without a new handshake. For session resumption, before calling wolfSSL_shutdown() with your session object, an application should save the session ID from the object with a call to wolfSSL_get1_session(), which returns a pointer to the session. Later, the application should create a new WOLFSSL object and assign the saved session with wolfSSL_set_session(). At this point, the application may call wolfSSL_connect() and wolfSSL will try to resume the session. The wolfSSL server code allows session resumption by default. The object returned by wolfSSL_get1_session() needs to be freed after the application is done with it by calling wolfSSL_SESSION_free() on it.

Parameters:

- **ssl** WOLFSSL structure to get session from.

See:

- wolfSSL_new
- wolfSSL_free
- wolfSSL_SESSION_free

Return:

- WOLFSSL_SESSION On success return session pointer.
- NULL will be returned if ssl is NULL, the SSL session cache is disabled, wolfSSL doesn't have the Session ID available, or mutex functions fail.

Example

```
WOLFSSL* ssl;
WOLFSSL_SESSION* ses;
// attempt/complete handshake
wolfSSL_connect(ssl);
ses  = wolfSSL_get1_session(ssl);
// check ses information
// disconnect / setup new SSL instance
wolfSSL_set_session(ssl, ses);
// attempt/resume handshake
wolfSSL_SESSION_free(ses);
```

C.58.4.137 function wolfSSLv23_client_method

```
WOLFSSL_METHOD * wolfSSLv23_client_method(
    void
)
```

The wolfSSLv23_client_method() function is used to indicate that the application is a client and will support the highest protocol version supported by the server between SSL 3.0 _ TLS 1.3. This function allocates memory for and initializes a new WOLFSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). Both wolfSSL clients and servers have robust version downgrade capability. If a specific protocol version method is used on either side, then only that version will be negotiated or an error will be returned. For example, a client that uses TLSv1 and tries to connect to a SSLv3 only server will fail, likewise connecting to a TLSv1.1 will fail as well. To resolve this issue, a client that uses the wolfSSLv23_client_method() function will use the highest protocol version supported by the server and downgrade to SSLv3 if needed. In this case, the client will be able to connect to a server running SSLv3 - TLSv1.3.

Parameters:

- **none** No parameters

See:

- wolfSSLv3_client_method
- wolfTLSv1_client_method
- wolfTLSv1_1_client_method
- wolfTLSv1_2_client_method
- wolfTLSv1_3_client_method
- wolfDTLSv1_client_method
- wolfSSL_CTX_new

Return:

- pointer upon success a pointer to a WOLFSSL_METHOD.
- Failure If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;
method = wolfSSLv23_client_method();
if (method == NULL) {
// unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.58.4.138 function wolfSSL_BIO_get_mem_data

```
int wolfSSL_BIO_get_mem_data(
    WOLFSSL_BIO * bio,
    void * p
)
```

This is used to set a byte pointer to the start of the internal memory buffer.

Parameters:

- **bio** WOLFSSL_BIO structure to get memory buffer of.

- **p** byte pointer to set to memory buffer.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_set_fp
- wolfSSL_BIO_free

Return:

- size On success the size of the buffer is returned
- SSL_FATAL_ERROR If an error case was encountered.

Example

```
WOLFSSL_BIO* bio;
const byte* p;
int ret;
bio  = wolfSSL_BIO_new(wolfSSL_BIO_s_mem());
ret  = wolfSSL_BIO_get_mem_data(bio, &p);
// check ret value
```

C.58.4.139 function wolfSSL_BIO_set_init

```
void wolfSSL_BIO_set_init(
    WOLFSSL_BIO * bio,
    int init
)
```

This is used to set the init flag of a BIO, indicating whether the BIO has been initialised and is ready for use. Typically called from a custom BIO create callback.

Parameters:

- **bio** WOLFSSL_BIO structure to set the init flag on.
- **init** value to set (0 = not initialised, 1 = initialised).

See:

- wolfSSL_BIO_get_init
- wolfSSL_BIO_new

Example

```
WOLFSSL_BIO* bio;
// inside a custom BIO create callback
wolfSSL_BIO_set_init(bio, 1);
```

C.58.4.140 function wolfSSL_BIO_get_init

```
int wolfSSL_BIO_get_init(
    WOLFSSL_BIO * bio
)
```

This is used to retrieve the init flag of a BIO, indicating whether the BIO has been initialised and is ready for use.

Parameters:

- **bio** WOLFSSL_BIO structure to query.

See:

- wolfSSL_BIO_set_init
- wolfSSL_BIO_new

Return:

- 1 if the BIO has been initialised.
- 0 if the BIO has not been initialised or bio is NULL.

Example

```
WOLFSSL_BIO* bio;
// create bio with custom method
if (wolfSSL_BIO_get_init(bio)) {
    // bio is ready
}
```

C.58.4.141 function wolfSSL_BIO_set_fd

```
long wolfSSL_BIO_set_fd(
    WOLFSSL_BIO * b,
    int fd,
    int flag
)
```

Sets the file descriptor for bio to use.

Parameters:

- **b** WOLFSSL_BIO structure to set fd.
- **fd** file descriptor to use.
- **flag** flag for behavior when closing fd.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_free

Return: SSL_SUCCESS(1) upon success.

Example

```
WOLFSSL_BIO* bio;
int fd;
// setup bio
wolfSSL_BIO_set_fd(bio, fd, BIO_NOCLOSE);
```

C.58.4.142 function wolfSSL_BIO_set_close

```
int wolfSSL_BIO_set_close(
    WOLFSSL_BIO * b,
    long flag
)
```

Sets the close flag, used to indicate that the i/o stream should be closed when the BIO is freed.

Parameters:

- **b** WOLFSSL_BIO structure.
- **flag** flag for behavior when closing i/o stream.

See:

- wolfSSL_BIO_new

- wolfSSL_BIO_free

Return: SSL_SUCCESS(1) upon success.

Example

```
WOLFSSL_BIO* bio;
// setup bio
wolfSSL_BIO_set_close(bio, BIO_NOCLOSE);
```

C.58.4.143 function wolfSSL_BIO_s_socket

```
WOLFSSL_BIO_METHOD * wolfSSL_BIO_s_socket(
    void
)
```

This is used to get a BIO_SOCKET type WOLFSSL_BIO_METHOD.

Parameters:

- **none** No parameters.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem

Return: WOLFSSL_BIO_METHOD pointer to a WOLFSSL_BIO_METHOD structure that is a socket type

Example

```
WOLFSSL_BIO* bio;
bio = wolfSSL_BIO_new(wolfSSL_BIO_s_socket);
```

C.58.4.144 function wolfSSL_BIO_set_write_buf_size

```
int wolfSSL_BIO_set_write_buf_size(
    WOLFSSL_BIO * b,
    long size
)
```

This is used to set the size of write buffer for a WOLFSSL_BIO. If write buffer has been previously set this function will free it when resetting the size. It is similar to wolfSSL_BIO_reset in that it resets read and write indexes to 0.

Parameters:

- **b** WOLFSSL_BIO structure to set write buffer size.
- **size** size of buffer to allocate.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_free

Return:

- SSL_SUCCESS On successfully setting the write buffer.
- SSL_FAILURE If an error case was encountered.

Example

```
WOLFSSL_BIO* bio;
int ret;
bio = wolfSSL_BIO_new(wolfSSL_BIO_s_mem());
ret = wolfSSL_BIO_set_write_buf_size(bio, 15000);
// check return value
```

C.58.4.145 function wolfSSL_BIO_make_bio_pair

```
int wolfSSL_BIO_make_bio_pair(
    WOLFSSL_BIO * b1,
    WOLFSSL_BIO * b2
)
```

This is used to pair two bios together. A pair of bios acts similar to a two way pipe writing to one can be read by the other and vice versa. It is expected that both bios be in the same thread, this function is not thread safe. Freeing one of the two bios removes both from being paired. If a write buffer size was not previously set for either of the bios it is set to a default size of 17000 (WOLFSSL_BIO_SIZE) before being paired.

Parameters:

- **b1** WOLFSSL_BIO structure to set pair.
- **b2** second WOLFSSL_BIO structure to complete pair.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_free

Return:

- SSL_SUCCESS On successfully pairing the two bios.
- SSL_FAILURE If an error case was encountered.

Example

```
WOLFSSL_BIO* bio;
WOLFSSL_BIO* bio2;
int ret;
bio  = wolfSSL_BIO_new(wolfSSL_BIO_s_bio());
bio2 = wolfSSL_BIO_new(wolfSSL_BIO_s_bio());
ret = wolfSSL_BIO_make_bio_pair(bio, bio2);
// check ret value
```

C.58.4.146 function wolfSSL_BIO_ctrl_reset_read_request

```
int wolfSSL_BIO_ctrl_reset_read_request(
    WOLFSSL_BIO * b
)
```

This is used to set the read request flag back to 0.

Parameters:

- **b** WOLFSSL_BIO structure to set read request flag.

See:

- wolfSSL_BIO_new, wolfSSL_BIO_s_mem
- wolfSSL_BIO_new, wolfSSL_BIO_free

Return:

- SSL_SUCCESS On successfully setting value.
- SSL_FAILURE If an error case was encountered.

Example

```
WOLFSSL_BIO* bio;
int ret;
...
ret = wolfSSL_BIO_ctrl_reset_read_request(bio);
// check ret value
```

C.58.4.147 function wolfSSL_BIO_nread0

```
int wolfSSL_BIO_nread0(
    WOLFSSL_BIO * bio,
    char ** buf
)
```

This is used to get a buffer pointer for reading from. Unlike wolfSSL_BIO_nread the internal read index is not advanced by the number returned from the function call. Reading past the value returned can result in reading out of array bounds.

Parameters:

- **bio** WOLFSSL_BIO structure to read from.
- **buf** pointer to set at beginning of read array.

See: wolfSSL_BIO_new

Return: >=0 on success return the number of bytes to read

Example

```
WOLFSSL_BIO* bio;
char* bufPt;
int ret;
// set up bio
ret = wolfSSL_BIO_nread0(bio, &bufPt); // read as many bytes as possible
// handle negative ret check
// read ret bytes from bufPt
```

C.58.4.148 function wolfSSL_BIO_nread

```
int wolfSSL_BIO_nread(
    WOLFSSL_BIO * bio,
    char ** buf,
    int num
)
```

This is used to get a buffer pointer for reading from. The internal read index is advanced by the number returned from the function call with buf being pointed to the beginning of the buffer to read from. In the case that less bytes are in the read buffer than the value requested with num the lesser value is returned. Reading past the value returned can result in reading out of array bounds.

Parameters:

- **bio** WOLFSSL_BIO structure to read from.
- **buf** pointer to set at beginning of read array.
- **num** number of bytes to try and read.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_nwrite

Return:

- =0 on success return the number of bytes to read
- WOLFSSL_BIO_ERROR(-1) on error case with nothing to read return -1

Example

```
WOLFSSL_BIO* bio;
char* bufPt;
int ret;

// set up bio
ret = wolfSSL_BIO_nread(bio, &bufPt, 10); // try to read 10 bytes
// handle negative ret check
// read ret bytes from bufPt
```

C.58.4.149 function wolfSSL_BIO_nwrite

```
int wolfSSL_BIO_nwrite(
    WOLFSSL_BIO * bio,
    char ** buf,
    int num
)
```

Gets a pointer to the buffer for writing as many bytes as returned by the function. Writing more bytes to the pointer returned then the value returned can result in writing out of bounds.

Parameters:

- **bio** WOLFSSL_BIO structure to write to.
- **buf** pointer to buffer to write to.
- **num** number of bytes desired to be written.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_free
- wolfSSL_BIO_nread

Return:

- int Returns the number of bytes that can be written to the buffer pointer returned.
- WOLFSSL_BIO_UNSET(-2) in the case that is not part of a bio pair
- WOLFSSL_BIO_ERROR(-1) in the case that there is no more room to write to

Example

```
WOLFSSL_BIO* bio;
char* bufPt;
int ret;
// set up bio
ret = wolfSSL_BIO_nwrite(bio, &bufPt, 10); // try to write 10 bytes
// handle negative ret check
// write ret bytes to bufPt
```

C.58.4.150 function wolfSSL_BIO_reset

```
int wolfSSL_BIO_reset(
    WOLFSSL_BIO * bio
)
```

Resets bio to an initial state. As an example for type BIO_BIO this resets the read and write index.

Parameters:

- **bio** WOLFSSL_BIO structure to reset.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_free

Return:

- 0 On successfully resetting the bio.
- WOLFSSL_BIO_ERROR(-1) Returned on bad input or unsuccessful reset.

Example

```
WOLFSSL_BIO* bio;
// setup bio
wolfSSL_BIO_reset(bio);
//use pt
```

C.58.4.151 function wolfSSL_BIO_seek

```
int wolfSSL_BIO_seek(
    WOLFSSL_BIO * bio,
    int ofs
)
```

This function adjusts the file pointer to the offset given. This is the offset from the head of the file.

Parameters:

- **bio** WOLFSSL_BIO structure to set.
- **ofs** offset into file.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_set_fp
- wolfSSL_BIO_free

Return:

- 0 On successfully seeking.
- -1 If an error case was encountered.

Example

```
WOLFSSL_BIO* bio;
XFILE fp;
int ret;
bio  = wolfSSL_BIO_new(wolfSSL_BIO_s_file());
ret  = wolfSSL_BIO_set_fp(bio, &fp);
// check ret value
```

```
ret  = wolfSSL_BIO_seek(bio, 3);
// check ret value
```

C.58.4.152 function wolfSSL_BIO_write_filename

```
int wolfSSL_BIO_write_filename(
    WOLFSSL_BIO * bio,
    char * name
)
```

This is used to set and write to a file. WIll overwrite any data currently in the file and is set to close the file when the bio is freed.

Parameters:

- **bio** WOLFSSL_BIO structure to set file.
- **name** name of file to write to.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_file
- wolfSSL_BIO_set_fp
- wolfSSL_BIO_free

Return:

- SSL_SUCCESS On successfully opening and setting file.
- SSL_FAILURE If an error case was encountered.

Example

```
WOLFSSL_BIO* bio;
int ret;
bio  = wolfSSL_BIO_new(wolfSSL_BIO_s_file());
ret  = wolfSSL_BIO_write_filename(bio, “test.txt”);
// check ret value
```

C.58.4.153 function wolfSSL_BIO_set_mem_eof_return

```
long wolfSSL_BIO_set_mem_eof_return(
    WOLFSSL_BIO * bio,
    int v
)
```

This is used to set the end of file value. Common value is -1 so as not to get confused with expected positive values.

Parameters:

- **bio** WOLFSSL_BIO structure to set end of file value.
- **v** value to set in bio.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_set_fp
- wolfSSL_BIO_free

Return: 0 returned on completion

Example

```
WOLFSSL_BIO* bio;
int ret;
bio  = wolfSSL_BIO_new(wolfSSL_BIO_s_mem());
ret  = wolfSSL_BIO_set_mem_eof_return(bio, -1);
// check ret value
```

C.58.4.154 function wolfSSL_BIO_get_mem_ptr

```
long wolfSSL_BIO_get_mem_ptr(
    WOLFSSL_BIO * bio,
    WOLFSSL_BUF_MEM ** m
)
```

This is a getter function for WOLFSSL_BIO memory pointer.

Parameters:

- **bio** pointer to the WOLFSSL_BIO structure for getting memory pointer.
- **m** pointer to WOLFSSL_BUF_MEM structure. Is set to point to bio's memory.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem

Return:

- SSL_SUCCESS On successfully getting the pointer SSL_SUCCESS is returned (currently value of 1).
- SSL_FAILURE Returned if NULL arguments are passed in (currently value of 0).

Example

```
WOLFSSL_BIO* bio;
WOLFSSL_BUF_MEM* pt;
// setup bio
wolfSSL_BIO_get_mem_ptr(bio, &pt);
//use pt
```

C.58.4.155 function wolfSSL_X509_NAME_oneline

```
char * wolfSSL_X509_NAME_oneline(
    WOLFSSL_X509_NAME * name,
    char * in,
    int sz
)
```

This function copies the name of the x509 into a buffer.

Parameters:

- **name** a pointer to a WOLFSSL_X509 structure.
- **in** a buffer to hold the name copied from the WOLFSSL_X509_NAME structure.
- **sz** the maximum size of the buffer.

See:

- wolfSSL_X509_get_subject_name
- wolfSSL_X509_get_issuer_name

- wolfSSL_X509_get_isCA
- wolfSSL_get_peer_certificate
- wolfSSL_X509_version

Return: A char pointer to the buffer with the WOLFSSL_X509_NAME structures name member's data is returned if the function executed normally.

Example

```
WOLFSSL_X509 x509;
char* name;
...
name = wolfSSL_X509_NAME_oneline(wolfSSL_X509_get_issuer_name(x509), 0, 0);

if(name <= 0){
    // There's nothing in the buffer.
}
```

C.58.4.156 function wolfSSL_X509_get_issuer_name

```
WOLFSSL_X509_NAME * wolfSSL_X509_get_issuer_name(
    WOLFSSL_X509 * cert
)
```

This function returns the name of the certificate issuer.

Parameters:

- **cert** a pointer to a WOLFSSL_X509 structure.

See:

- wolfSSL_X509_get_subject_name
- wolfSSL_X509_get_isCA
- wolfSSL_get_peer_certificate
- wolfSSL_X509_NAME_oneline

Return:

- point a pointer to the WOLFSSL_X509 struct's issuer member is returned.
- NULL if the cert passed in is NULL.

Example

```
WOLFSSL_X509* x509;
WOLFSSL_X509_NAME issuer;
...
issuer = wolfSSL_X509_NAME_oneline(wolfSSL_X509_get_issuer_name(x509), 0, 0);

if(!issuer){
    // NULL was returned
} else {
    // issuer hods the name of the certificate issuer.
}
```

C.58.4.157 function wolfSSL_X509_get_subject_name

```
WOLFSSL_X509_NAME * wolfSSL_X509_get_subject_name(
    WOLFSSL_X509 * cert
)
```

This function returns the subject member of the WOLFSSL_X509 structure.

Parameters:

- **cert** a pointer to a WOLFSSL_X509 structure.

See:

- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_isCA
- wolfSSL_get_peer_certificate

Return: pointer a pointer to the WOLFSSL_X509_NAME structure. The pointer may be NULL if the WOLFSSL_X509 struct is NULL or if the subject member of the structure is NULL.

Example

```
WOLFSSL_X509* cert;
WOLFSSL_X509_NAME name;
…
name = wolfSSL_X509_get_subject_name(cert);
if(name == NULL){
    // Deal with the NULL cacse
}
```

C.58.4.158 function wolfSSL_X509_get_isCA

```
int wolfSSL_X509_get_isCA(
    WOLFSSL_X509 * x509
)
```

Checks the isCa member of the WOLFSSL_X509 structure and returns the value.

Parameters:

- **x509** a pointer to a WOLFSSL_X509 structure.

See:

- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_isCA

Return:

- isCA returns the value in the isCA member of the WOLFSSL_X509 structure is returned.
- 0 returned if there is not a valid x509 structure passed in.

Example

```
WOLFSSL* ssl;
...
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_X509_get_isCA(ssl)){
    // This is the CA
}else {
    // Failure case
}
```

C.58.4.159 function wolfSSL_X509_NAME_get_text_by_NID

```
int wolfSSL_X509_NAME_get_text_by_NID(
    const WOLFSSL_X509_NAME * name,
    int nid,
    char * buf,
    int len
)
```

This function gets the text related to the passed in NID value.

Parameters:

- **name** WOLFSSL_X509_NAME to search for text.
- **nid** NID to search for.
- **buf** buffer to hold text when found.
- **len** length of buffer.

See: none

Return: int returns the size of the text buffer.

Example

```
WOLFSSL_X509_NAME* name;
char buffer[100];
int bufferSz;
int ret;
// get WOLFSSL_X509_NAME
ret = wolfSSL_X509_NAME_get_text_by_NID(name, NID_commonName,
buffer, bufferSz);

//check ret value
```

C.58.4.160 function wolfSSL_X509_get_signature_type

```
int wolfSSL_X509_get_signature_type(
    WOLFSSL_X509 * x509
)
```

This function returns the value stored in the sigOID member of the WOLFSSL_X509 structure.

Parameters:

- **x509** a pointer to a WOLFSSL_X509 structure.

See:

- wolfSSL_X509_get_signature
- wolfSSL_X509_version
- wolfSSL_X509_get_der
- wolfSSL_X509_get_serial_number
- wolfSSL_X509_notBefore
- wolfSSL_X509_notAfter
- wolfSSL_X509_free

Return:

- 0 returned if the WOLFSSL_X509 structure is NULL.
- int an integer value is returned which was retrieved from the x509 object.

Example

```
WOLFSSL_X509 x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                        DYNAMIC_TYPE_X509);
...
int x509SigType = wolfSSL_X509_get_signature_type(x509);

if(x509SigType != EXPECTED){
// Deal with an unexpected value
}
```

C.58.4.161 function wolfSSL_X509_free

```
void wolfSSL_X509_free(
    WOLFSSL_X509 * x509
)
```

This function frees a WOLFSSL_X509 structure.

Parameters:

- **x509** a pointer to the WOLFSSL_X509 struct.

See:

- wolfSSL_X509_get_signature
- wolfSSL_X509_version
- wolfSSL_X509_get_der
- wolfSSL_X509_get_serial_number
- wolfSSL_X509_notBefore
- wolfSSL_X509_notAfter

Example

```
WOLFSSL_X509* x509 = (WOLFSSL_X509*)XMALOC(sizeof(WOLFSSL_X509), NULL,
DYNAMIC_TYPE_X509) ;

wolfSSL_X509_free(x509);
```

C.58.4.162 function wolfSSL_X509_get_signature

```
int wolfSSL_X509_get_signature(
    WOLFSSL_X509 * x509,
    unsigned char * buf,
    int * bufSz
)
```

Gets the X509 signature and stores it in the buffer.

Parameters:

- **x509** pointer to a WOLFSSL_X509 structure.
- **buf** a char pointer to the buffer.
- **bufSz** an integer pointer to the size of the buffer.

See:

- wolfSSL_X509_get_serial_number
- wolfSSL_X509_get_signature_type
- wolfSSL_X509_get_device_type

Return:

- SSL_SUCCESS returned if the function successfully executes. The signature is loaded into the buffer.
- SSL_FATAL_ERRROR returns if the x509 struct or the bufSz member is NULL. There is also a check for the length member of the sig structure (sig is a member of x509).

Example

```
WOLFSSL_X509* x509 = (WOLFSSL_X509)XMALOC(sizeof(WOLFSSL_X509), NULL,
DYNAMIC_TYPE_X509);
unsigned char* buf; // Initialize
int* bufSz = sizeof(buf)/sizeof(unsigned char);
...
if(wolfSSL_X509_get_signature(x509, buf, bufSz) != SSL_SUCCESS){
    // The function did not execute successfully.
} else{
    // The buffer was written to correctly.
}
```

C.58.4.163 function wolfSSL_X509_STORE_add_cert

```
int wolfSSL_X509_STORE_add_cert(
    WOLFSSL_X509_STORE * store,
    WOLFSSL_X509 * x509
)
```

This function adds a certificate to the WOLFSSL_X509_STRE structure.

Parameters:

- **store** certificate store to add the certificate to.
- **x509** certificate to add.

See: wolfSSL_X509_free

Return:

- SSL_SUCCESS If certificate is added successfully.
- SSL_FATAL_ERROR: If certificate is not added successfully.

Example

```
WOLFSSL_X509_STORE* str;
WOLFSSL_X509* x509;
int ret;
ret = wolfSSL_X509_STORE_add_cert(str, x509);
//check ret value
```

C.58.4.164 function wolfSSL_X509_STORE_CTX_get_chain

```
WOLFSSL_STACK * wolfSSL_X509_STORE_CTX_get_chain(
    WOLFSSL_X509_STORE_CTX * ctx
)
```

This function is a getter function for chain variable in WOLFSSL_X509_STORE_CTX structure. Currently chain is not populated.

Parameters:

- **ctx** certificate store ctx to get parse chain from.

See: wolfSSL_sk_X509_free

Return:

- pointer if successful returns WOLFSSL_STACK (same as STACK_OF(WOLFSSL_X509)) pointer
- Null upon failure

Example

```
WOLFSSL_STACK* sk;
WOLFSSL_X509_STORE_CTX* ctx;
sk = wolfSSL_X509_STORE_CTX_get_chain(ctx);
//check sk for NULL and then use it. sk needs freed after done.
```

C.58.4.165 function wolfSSL_X509_STORE_set_flags

```
int wolfSSL_X509_STORE_set_flags(
    WOLFSSL_X509_STORE * store,
    unsigned long flag
)
```

This function takes in a flag to change the behavior of the WOLFSSL_X509_STORE structure passed in. An example of a flag used is WOLFSSL_CRL_CHECK. X509_V_FLAG_CRL_CHECK and X509_V_FLAG_CRL_CHECK_ALL enable CRL checking like WOLFSSL_CRL_CHECK and WOLFSSL_CRL_CHECKALL.

Parameters:

- **str** certificate store to set flag in.
- **flag** flag for behavior.

See:

- wolfSSL_X509_STORE_new
- wolfSSL_X509_STORE_free

Return:

- SSL_SUCCESS If no errors were encountered when setting the flag.
- <0 a negative value will be returned upon failure.

Example

```
WOLFSSL_X509_STORE* str;
int ret;
// create and set up str
ret = wolfSSL_X509_STORE_set_flags(str, WOLFSSL_CRL_CHECKALL);
If (ret != SSL_SUCCESS) {
    //check ret value and handle error case
}
```

C.58.4.166 function wolfSSL_X509_notBefore

```
const byte * wolfSSL_X509_notBefore(
    WOLFSSL_X509 * x509
)
```

This function the certificate “not before” validity encoded as a byte array.

Parameters:

- **x509** pointer to a WOLFSSL_X509 structure.

See:

- wolfSSL_X509_get_signature
- wolfSSL_X509_version
- wolfSSL_X509_get_der
- wolfSSL_X509_get_serial_number
- wolfSSL_X509_notAfter
- wolfSSL_X509_free

Return:

- NULL returned if the WOLFSSL_X509 structure is NULL.
- byte is returned that contains the notBeforeData.

Example

```
WOLFSSL_X509* x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                        DYNAMIC_TYPE_X509);
...
byte* notBeforeData = wolfSSL_X509_notBefore(x509);
```

C.58.4.167 function wolfSSL_X509_notAfter

```
const byte * wolfSSL_X509_notAfter(
    WOLFSSL_X509 * x509
)
```

This function the certificate “not after” validity encoded as a byte array.

Parameters:

- **x509** pointer to a WOLFSSL_X509 structure.

See:

- wolfSSL_X509_get_signature
- wolfSSL_X509_version
- wolfSSL_X509_get_der
- wolfSSL_X509_get_serial_number
- wolfSSL_X509_notBefore
- wolfSSL_X509_free

Return:

- NULL returned if the WOLFSSL_X509 structure is NULL.
- byte is returned that contains the notAfterData.

Example

```
WOLFSSL_X509* x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                        DYNAMIC_TYPE_X509);
...
byte* notAfterData = wolfSSL_X509_notAfter(x509);
```

C.58.4.168 function wolfSSL_ASN1_INTEGER_to_BN

```
WOLFSSL_BIGNUM * wolfSSL_ASN1_INTEGER_to_BN(
    const WOLFSSL_ASN1_INTEGER * ai,
    WOLFSSL_BIGNUM * bn
)
```

This function is used to copy a WOLFSSL_ASN1_INTEGER value to a WOLFSSL_BIGNUM structure.

Parameters:

- **ai** WOLFSSL_ASN1_INTEGER structure to copy from.
- **bn** if wanting to copy into an already existing WOLFSSL_BIGNUM struct then pass in a pointer to it. Optionally this can be NULL and a new WOLFSSL_BIGNUM structure will be created.

See: none

Return:

- pointer On successfully copying the WOLFSSL_ASN1_INTEGER value a WOLFSSL_BIGNUM pointer is returned.
- Null upon failure.

Example

```
WOLFSSL_ASN1_INTEGER* ai;
WOLFSSL_BIGNUM* bn;
// create ai
bn = wolfSSL_ASN1_INTEGER_to_BN(ai, NULL);

// or if having already created bn and wanting to reuse structure
// wolfSSL_ASN1_INTEGER_to_BN(ai, bn);
// check bn is or return value is not NULL
```

C.58.4.169 function wolfSSL_CTX_add_extra_chain_cert

```
long wolfSSL_CTX_add_extra_chain_cert(
    WOLFSSL_CTX * ctx,
    WOLFSSL_X509 * x509
)
```

This function adds the certificate to the internal chain being built in the WOLFSSL_CTX structure.

Parameters:

- **ctx** WOLFSSL_CTX structure to add certificate to.
- **x509** certificate to add to the chain.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free

Return:

- SSL_SUCCESS after successfully adding the certificate.
- SSL_FAILURE if failing to add the certificate to the chain.

Example

```
WOLFSSL_CTX* ctx;
WOLFSSL_X509* x509;
int ret;
// create ctx
ret = wolfSSL_CTX_add_extra_chain_cert(ctx, x509);
// check ret value
```

C.58.4.170 function wolfSSL_CTX_get_read_ahead

```
int wolfSSL_CTX_get_read_ahead(
    WOLFSSL_CTX * ctx
)
```

This function returns the get read ahead flag from a WOLFSSL_CTX structure.

Parameters:

- **ctx** WOLFSSL_CTX structure to get read ahead flag from.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free
- wolfSSL_CTX_set_read_ahead

Return:

- flag On success returns the read ahead flag.
- SSL_FAILURE If ctx is NULL then SSL_FAILURE is returned.

Example

```
WOLFSSL_CTX* ctx;
int flag;
// setup ctx
flag = wolfSSL_CTX_get_read_ahead(ctx);
//check flag
```

C.58.4.171 function wolfSSL_CTX_set_read_ahead

```
int wolfSSL_CTX_set_read_ahead(
    WOLFSSL_CTX * ctx,
    int v
)
```

This function sets the read ahead flag in the WOLFSSL_CTX structure. When enabled, the record-header read pulls in up to a full TLS record in a single recv(), so the body (and any following buffered records) is obtained without a second syscall. The flag only changes I/O behaviour when the library is built with read-ahead support (–enable-readahead / WOLFSSL_TLS_READ_AHEAD); otherwise it is stored but inert.

Parameters:

- **ctx** WOLFSSL_CTX structure to set read ahead flag.
- **v** read ahead flag

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free
- wolfSSL_CTX_get_read_ahead
- wolfSSL_has_pending

Return:

- SSL_SUCCESS If ctx read ahead flag set.
- SSL_FAILURE If ctx is NULL then SSL_FAILURE is returned.

Note: With read-ahead enabled, undecrypted data can remain buffered internally while the socket has no more data to read. wolfSSL_has_pending() reports whether any such data is buffered, but a non-zero return does not guarantee a full record is available: wolfSSL_read() may still return WANT_READ when only a partial record is buffered. Event_driven applications should therefore drain the connection by calling wolfSSL_read() until it returns WANT_READ (or an error) before returning to select()/poll(), rather than looping on wolfSSL_has_pending() alone; otherwise buffered data could be missed or the loop could spin.

Example

```
WOLFSSL_CTX* ctx;
int flag;
int ret;
// setup ctx
ret = wolfSSL_CTX_set_read_ahead(ctx, flag);
// check return value
```

C.58.4.172 function wolfSSL_CTX_set_default_read_buffer_len

```
int wolfSSL_CTX_set_default_read_buffer_len(
    WOLFSSL_CTX * ctx,
    size_t len
)
```

This function sets the read-ahead receive window size for contexts created from this WOLFSSL_CTX. When read-ahead is enabled (wolfSSL_CTX_set_read_ahead()), a single recv() pulls in up to `len` bytes:

Parameters:

- **ctx** WOLFSSL_CTX structure to set the read-ahead buffer length on.
- **len** read-ahead coalescing buffer size in bytes (0 = one record).

See:

- wolfSSL_CTX_set_read_ahead
- wolfSSL_set_default_read_buffer_len
- wolfSSL_has_pending

Return:

- SSL_SUCCESS If the buffer length was set.
- SSL_FAILURE If ctx is NULL.

Note: This is a memory-vs-syscall trade-off. While read-ahead is enabled the input buffer is retained (bounded to `len`) for the connection's lifetime, so a large `len` multiplied by many concurrent connections is persistent memory, while a small `len` bounds per-connection footprint at the cost of more syscalls for records that exceed it.

- `len` of 0 resets the window to the one-record default. This is also the window a freshly created context already carries, so the record body is read together with its header without a second syscall.
- A `len` larger than one record lets a single recv() coalesce several back-to-back records, one syscall instead of one per record.
- A `len` smaller than one record caps the receive buffer's footprint when the peer's records are known to be small (e.g. 4 KB), saving heap versus the one-record default. `len` is only a speculative read window, not a hard limit: a record larger than `len` is still received correctly, with the input buffer grown to the record's actual size on demand (costing an extra syscall and reallocation for that record) and then reallocated back down to the window once the oversized record is consumed, so the retained footprint stays bounded by `len` rather than by the largest record seen. The setting only affects I/O when the library is built with read-ahead support (–enable-readahead / WOLFSSL_TLS_READ_AHEAD); otherwise it is stored but inert. This is the wolfSSL equivalent of OpenSSL's SSL_CTX_set_default_read_buffer_len(); unlike OpenSSL it returns a status code (callers that ignore the return remain source-compatible) and it honours sizes below one record, whereas OpenSSL only ever enlarges the buffer. A `len` above WOLFSSL_MAX_READ_AHEAD_SZ (16 MB) is clamped to that maximum. Because 0 and oversized values are both normalised, wolfSSL_CTX_get_default_read_buffer_len() reports the effective window (the one_record default rather than 0 when unset), not the raw argument.

Example

```
WOLFSSL_CTX* ctx;
// setup ctx
wolfSSL_CTX_set_read_ahead(ctx, 1);
// coalesce up to four max-size records per recv()
wolfSSL_CTX_set_default_read_buffer_len(ctx, 4 * 16384);
```

C.58.4.173 function wolfSSL_set_default_read_buffer_len

```
int wolfSSL_set_default_read_buffer_len(
    WOLFSSL * ssl,
    size_t len
)
```

This function sets the read-ahead coalescing buffer size on a single WOLFSSL session, overriding the value inherited from its WOLFSSL_CTX. See wolfSSL_CTX_set_default_read_buffer_len() for the full description, the one-record default, and the memory-vs-syscall trade-off.

Parameters:

- **ssl** WOLFSSL structure to set the read-ahead buffer length on.
- **len** read-ahead coalescing buffer size in bytes (0 = one record).

See:

- wolfSSL_CTX_set_default_read_buffer_len
- wolfSSL_set_read_ahead
- wolfSSL_has_pending

Return:

- SSL_SUCCESS If the buffer length was set.
- SSL_FAILURE If ssl is NULL.

C.58.4.174 function wolfSSL_CTX_get_default_read_buffer_len

```
long wolfSSL_CTX_get_default_read_buffer_len(
    WOLFSSL_CTX * ctx
)
```

This function returns the read-ahead coalescing buffer size configured on a WOLFSSL_CTX by wolfSSL_CTX_set_default_read_buffer_len(). Because a length of 0 and oversized values are normalised when set, the value reported is the effective window actually in use (the one_record default rather than 0 when the length was never changed), not the raw argument last passed.

Parameters:

- **ctx** WOLFSSL_CTX structure to get the read-ahead buffer length from.

See:

- wolfSSL_CTX_set_default_read_buffer_len
- wolfSSL_get_default_read_buffer_len
- wolfSSL_CTX_set_read_ahead

Return:

- len On success returns the read-ahead buffer length in bytes.
- SSL_FAILURE If ctx is NULL.

Example

```
WOLFSSL_CTX* ctx;
long len;
// setup ctx
len = wolfSSL_CTX_get_default_read_buffer_len(ctx);
// check len
```

C.58.4.175 function wolfSSL_get_default_read_buffer_len

```
long wolfSSL_get_default_read_buffer_len(
    const WOLFSSL * ssl
)
```

This function returns the read-ahead coalescing buffer size in effect for a single WOLFSSL session, whether inherited from its WOLFSSL_CTX or overridden by wolfSSL_set_default_read_buffer_len(). As with the WOLFSSL_CTX getter, the value reported is the effective window in use, not the raw argument last passed.

Parameters:

- **ssl** WOLFSSL structure to get the read-ahead buffer length from.

See:

- wolfSSL_set_default_read_buffer_len
- wolfSSL_CTX_get_default_read_buffer_len
- wolfSSL_set_read_ahead

Return:

- len On success returns the read-ahead buffer length in bytes.
- SSL_FAILURE If ssl is NULL.

Example

```
WOLFSSL* ssl;
long len;
// setup ssl
len = wolfSSL_get_default_read_buffer_len(ssl);
// check len
```

C.58.4.176 function wolfSSL_CTX_set_max_send_fragment

```
int wolfSSL_CTX_set_max_send_fragment(
    WOLFSSL_CTX * ctx,
    long max_fragment
)
```

This function caps the plaintext carried in each record sent by WOLFSSL objects created from the context afterwards. It is the wolfSSL equivalent of OpenSSL's SSL_CTX_set_max_send_fragment(). The cap is local and send only: the peer is not told and records received are not limited. Unset, there is no cap. wolfSSL_new() copies the value, so a later call does not change existing objects, as in OpenSSL. With a negotiated max_fragment_length the smaller of the two limits applies, where OpenSSL uses the negotiated length instead. A DTLS write larger than the cap fails with DTLS_SIZE_ERROR, like one larger than the MTU, unless WOLFSSL_NO_DTLS_SIZE_CHECK is defined. In TLS 1.3 only Certificate and CertificateVerify are split to the cap. Other handshake messages, such as ClientHello, ServerHello, EncryptedExtensions, CertificateRequest and NewSessionTicket, are sent in one record each, as they are with a negotiated max_fragment_length. Application data is always split.

Parameters:

- **ctx** WOLFSSL_CTX structure to set the cap on.
- **max_fragment** most plaintext bytes in a record sent.

See:

- wolfSSL_set_max_send_fragment
- wolfSSL_CTX_UseMaxFragment
- wolfSSL_GetMaxOutputSize

Return:

- WOLFSSL_SUCCESS (1) If the cap was set.
- WOLFSSL_FAILURE (0) If ctx is NULL or max_fragment is not in the range 512 to 16384.

Example

```
WOLFSSL_CTX* ctx;
// setup ctx
if (wolfSSL_CTX_set_max_send_fragment(ctx, 1024) != WOLFSSL_SUCCESS) {
    // value out of range
}
```

C.58.4.177 function wolfSSL_set_max_send_fragment

```
int wolfSSL_set_max_send_fragment(
    WOLFSSL * ssl,
    long max_fragment
)
```

This function caps the plaintext carried in each record sent by a single WOLFSSL object, replacing the value copied from its WOLFSSL_CTX. It is the wolfSSL equivalent of OpenSSL's SSL_set_max_send_fragment() and takes effect from the next record sent. See wolfSSL_CTX_set_max_send_fragment() for the full description.

Parameters:

- **ssl** WOLFSSL structure to set the cap on.
- **max_fragment** most plaintext bytes in a record sent.

See:

- wolfSSL_CTX_set_max_send_fragment
- wolfSSL_UseMaxFragment
- wolfSSL_GetMaxOutputSize

Return:

- WOLFSSL_SUCCESS (1) If the cap was set.
- WOLFSSL_FAILURE (0) If ssl is NULL or max_fragment is not in the range 512 to 16384.

Example

```
WOLFSSL* ssl;
// setup ssl
if (wolfSSL_set_max_send_fragment(ssl, 512) != WOLFSSL_SUCCESS) {
    // value out of range
}
```

C.58.4.178 function wolfSSL_CTX_set_tlsext_status_arg

```
long wolfSSL_CTX_set_tlsext_status_arg(
    WOLFSSL_CTX * ctx,
    void * arg
)
```

This function sets the argument to be passed to the OCSP status callback.

Parameters:

- **ctx** WOLFSSL_CTX structure to set user argument.
- **arg** user argument.

See:

- wolfSSL_CTX_set_tlsext_status_cb
- wolfSSL_CTX_new
- wolfSSL_CTX_free

Return:

- SSL_FAILURE If ctx or it's cert manager is NULL, or stapling is not set up.
- SSL_SUCCESS If successfully set.

Example

```
WOLFSSL_CTX* ctx;
void* data;
int ret;
// setup ctx
ret = wolfSSL_CTX_set_tlsext_status_arg(ctx, data);

//check ret value
```

C.58.4.179 function wolfSSL_CTX_set_client_cert_cb

```
void wolfSSL_CTX_set_client_cert_cb(
    WOLFSSL_CTX * ctx,
    client_cert_cb cb
)
```

Sets a callback to select the client certificate and private key.

Parameters:

- **ctx** The WOLFSSL_CTX object.
- **cb** The callback function to select the client certificate and key.

See: wolfSSL_CTX_set_cert_cb

Return: void

This function allows the application to register a callback that will be invoked when a client certificate is requested during the handshake. The callback can select and provide the certificate and key to use.

Example

```
int my_client_cert_cb(WOLFSSL *ssl, WOLFSSL_X509 **x509, WOLFSSL_EVP_PKEY
 ↪  **pkey) { ... }
wolfSSL_CTX_set_client_cert_cb(ctx, my_client_cert_cb);
```

C.58.4.180 function wolfSSL_CTX_set_cert_cb

```
void wolfSSL_CTX_set_cert_cb(
    WOLFSSL_CTX * ctx,
    CertSetupCallback cb,
    void * arg
)
```

Sets a generic certificate setup callback.

Parameters:

- **ctx** The WOLFSSL_CTX object.
- **cb** The callback function for certificate setup.
- **arg** User argument to pass to the callback.

See:

- wolfSSL_CTX_set_client_cert_cb
- wolfSSL_get0_peer_CA_list
- wolfSSL_get_client_CA_list

Return: void

The callback is called whenever a certificate is about to be used, so the application can inspect, set or clear certificates - for example to react to a CA list sent by the peer.

Example

```
int my_cert_setup_cb(WOLFSSL* ssl, void* arg) { ... }
wolfSSL_CTX_set_cert_cb(ctx, my_cert_setup_cb, NULL);
```

C.58.4.181 function wolfSSL_CTX_set_tlsext_status_cb

```
int wolfSSL_CTX_set_tlsext_status_cb(
    WOLFSSL_CTX * ctx,
    tlsextStatusCb cb
)
```

Sets the callback to be used for handling OCSP status requests (OCSP stapling).

Parameters:

- **ctx** The WOLFSSL_CTX object.
- **cb** The callback function to handle OCSP status requests. NULL clears any callback already set.

See:

- wolfSSL_CTX_get_tlsext_status_cb
- wolfSSL_CTX_set_tlsext_status_arg

Return:

- SSL_SUCCESS on success, SSL_FAILURE otherwise.
- SSL_FAILURE will be returned when ctx or its certificate manager is NULL, or stapling is not set up. A NULL cb is not a failure.

This function allows the application to register a callback that will be invoked when an OCSP status request is received during the TLS handshake. The callback can provide an OCSP response to be stapled to the handshake. This API is only useful on the server side.

Example

```
int my_ocsp_status_cb(WOLFSSL* ssl, void* arg) { ... }
wolfSSL_CTX_set_tlsext_status_cb(ctx, my_ocsp_status_cb);
```

C.58.4.182 function wolfSSL_CTX_get_tlsext_status_cb

```
int wolfSSL_CTX_get_tlsext_status_cb(
    WOLFSSL_CTX * ctx,
    tlsextStatusCb * cb
)
```

Gets the currently set OCSP status callback for the context.

Parameters:

- **ctx** The WOLFSSL_CTX object.
- **cb** Pointer to receive the callback function. Required.

See: wolfSSL_CTX_set_tlsext_status_cb

Return:

- SSL_SUCCESS on success, SSL_FAILURE otherwise.
- SSL_FAILURE will be returned when ctx, its certificate manager or cb is NULL, or stapling is not set up.

C.58.4.183 function wolfSSL_get_tlsext_status_ocsp_resp

```
long wolfSSL_get_tlsext_status_ocsp_resp(
    WOLFSSL * ssl,
    unsigned char ** resp
)
```

Gets the OCSP response that will be sent (stapled) to the peer.

Parameters:

- **ssl** The WOLFSSL session.
- **resp** Pointer to receive the response buffer.

See: wolfSSL_set_tlsext_status_ocsp_resp

Return: Length of the response, or negative value on error.

C.58.4.184 function wolfSSL_set_tlsext_status_ocsp_resp

```
long wolfSSL_set_tlsext_status_ocsp_resp(
    WOLFSSL * ssl,
    unsigned char * resp,
    int len
)
```

Sets the OCSP response to be sent (stapled) to the peer.

Parameters:

- **ssl** The WOLFSSL session.
- **resp** Pointer to the response buffer.
- **len** Length of the response buffer.

See: wolfSSL_get_tlsext_status_ocsp_resp

Return:

- SSL_SUCCESS on success, SSL_FAILURE otherwise.
- SSL_FAILURE will be returned when ssl is NULL or the response and length disagree.

The buffer in resp becomes owned by wolfSSL and will be freed by wolfSSL. The application must not free the buffer after calling this function.

C.58.4.185 function wolfSSL_set_tlsext_status_ocsp_resp_multi

```
int wolfSSL_set_tlsext_status_ocsp_resp_multi(
    WOLFSSL * ssl,
    unsigned char * resp,
    int len,
    word32 idx
)
```

Sets multiple OCSP responses for TLS multi-certificate chains.

Parameters:

- **ssl** The WOLFSSL session.
- **resp** Pointer to the response buffer.
- **len** Length of the response buffer.
- **idx** Index of the certificate chain.

Return:

- SSL_SUCCESS on success, SSL_FAILURE otherwise.
- SSL_FAILURE will be returned when ssl is NULL, idx is out of range, or the response and length disagree.

The buffer in resp becomes owned by wolfSSL and will be freed by wolfSSL. The application must not free the buffer after calling this function.

Each stapled response (one per certificate) must carry exactly one SingleResponse. A BasicOCSPResponse bundling more than one SingleResponse is rejected by the peer, so responders must supply a separate response per certificate rather than combining statuses into a single response.

C.58.4.186 function wolfSSL_CTX_set_ocsp_status_verify_cb

```
void wolfSSL_CTX_set_ocsp_status_verify_cb(
    WOLFSSL_CTX * ctx,
    ocspVerifyStatusCb cb,
    void * cbArg
)
```

Sets a callback to verify the OCSP status response.

Parameters:

- **ctx** The WOLFSSL_CTX object.
- **cb** The callback function.
- **cbArg** User argument to pass to the callback.

Return: void

It is recommended to enable SESSION_CERTS in order to have access to the peer's certificate chain during OCSP verification.

Example

```
void my_ocsp_verify_cb(WOLFSSL* ssl, int err, byte* resp, word32 respSz, word32
 ↪  idx, void* arg)
{
    (void)arg;
    if (err == 0 && staple && stapleSz > 0) {
        printf("Client: OCSP staple received, size=%u\n", stapleSz);
        return 0;
    }
    // Manual OCSP staple verification if err != 0
    if (err != 0 && staple && stapleSz > 0) {
        WOLFSSL_CERT_MANAGER* cm = NULL;
        DecodedCert cert;
        byte certInit = 0;
        WOLFSSL_OCSP* ocsp = NULL;
        WOLFSSL_X509_CHAIN* peerCerts;
        int i;

        cm = wolfSSL_CertManagerNew();
        if (cm == NULL)
            goto cleanup;
        if (wolfSSL_CertManagerLoadCA(cm, CA_CERT, NULL) != WOLFSSL_SUCCESS)
            goto cleanup;

        peerCerts = wolfSSL_get_peer_chain(ssl);
        if (peerCerts == NULL || wolfSSL_get_chain_count(peerCerts) <= (int)idx
                || wolfSSL_get_chain_length(peerCerts, (int)idx) == 0)
            goto cleanup;

        for (i = idx + 1; i < wolfSSL_get_chain_count(peerCerts); i++) {
            if (wolfSSL_get_chain_length(peerCerts, i) == 0)
                continue;   // a certificate too large to store leaves its slot
                 ↪  empty
            if (wolfSSL_CertManagerLoadCABuffer(cm,
                 ↪  wolfSSL_get_chain_cert(peerCerts, i),
                    wolfSSL_get_chain_length(peerCerts, i),
 ↪  WOLFSSL_FILETYPE_ASN1) != WOLFSSL_SUCCESS)
                goto cleanup;
        }

        wc_InitDecodedCert(&cert, wolfSSL_get_chain_cert(peerCerts, idx),
 ↪  wolfSSL_get_chain_length(peerCerts, idx), NULL);
        certInit = 1;
        if (wc_ParseCert(&cert, CERT_TYPE, VERIFY, cm) != 0)
            goto cleanup;
        if ((ocsp = wc_NewOCSP(cm)) == NULL)
            goto cleanup;
        if (wc_CheckCertOcspResponse(ocsp, &cert, staple, stapleSz, NULL) != 0)
            goto cleanup;

        printf("Client: Manual OCSP staple verification succeeded for
 ↪  idx=%u\n", idx);
        err = 0;
cleanup:
        wc_FreeOCSP(ocsp);
```

```
        if (certInit)
            wc_FreeDecodedCert(&cert);
        wolfSSL_CertManagerFree(cm);
        if (err == 0)
            return 0;
        printf("Client: Manual OCSP staple verification failed for idx=%u\n",
 ↪  idx);
    }
    printf("Client: OCSP staple verify error=%d\n", err);
    return err;
}
wolfSSL_CTX_set_ocsp_status_verify_cb(ctx, my_ocsp_verify_cb, NULL);
```

C.58.4.187 function wolfSSL_CTX_set_tlsext_opaque_prf_input_callback_arg

```
long wolfSSL_CTX_set_tlsext_opaque_prf_input_callback_arg(
    WOLFSSL_CTX * ctx,
    void * arg
)
```

This function sets the optional argument to be passed to the PRF callback.

Parameters:

- **ctx** WOLFSSL_CTX structure to set user argument.
- **arg** user argument.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free

Return:

- SSL_FAILURE If ctx is NULL.
- SSL_SUCCESS If successfully set.

Example

```
WOLFSSL_CTX* ctx;
void* data;
int ret;
// setup ctx
ret = wolfSSL_CTX_set_tlsext_opaques_prf_input_callback_arg(ctx, data);
//check ret value
```

C.58.4.188 function wolfSSL_CTX_set_options

```
long wolfSSL_CTX_set_options(
    WOLFSSL_CTX * ctx,
    long opt
)
```

This function ORs the bits in opt into the options mask of the given WOLFSSL_CTX. The options mask is inherited by every WOLFSSL session later created from this context. Bits are accumulated — to remove an option, use wolfSSL_CTX_clear_options(). The OpenSSL-style "SSL_OP_" *macros are aliases for the corresponding "WOLFSSL_OP_"* values; either spelling may be used.

Parameters:

- **ctx** WOLFSSL_CTX structure on which to set the options mask.
- **opt** the bitmask of SSL_OP_* / WOLFSSL_OP_* flags to OR into the current mask.

See:

- wolfSSL_CTX_clear_options
- wolfSSL_CTX_get_options
- wolfSSL_set_options
- wolfSSL_get_options

Return:

- BAD_FUNC_ARG if `ctx` is NULL.
- The updated options mask value stored in `ctx` on success.

Effective options:

Macro	Effect
SSL_OP_NO_SSLv2	Disable SSLv2 (wolfSSL never supports SSLv2; flag is accepted for OpenSSL compatibility)
SSL_OP_NO_SSLv3	Disable SSLv3
SSL_OP_NO_TLSv1	Disable TLS 1.0
SSL_OP_NO_TLSv1_1	Disable TLS 1.1
SSL_OP_NO_TLSv1_2	Disable TLS 1.2
SSL_OP_NO_TLSv1_3	Disable TLS 1.3 (requires WOLFSSL_TLS13)
SSL_OP_NO_COMPRESSION	Disable record-layer compression (no-op unless HAVE_LIBZ)
SSL_OP_NO_TICKET	Disable RFC 5077 session tickets (TLS 1.2 only; TLS 1.3 ignores this flag); requires HAVE_SESSION_TICKET and (OPENSSL_EXTRA or HAVE_WEBSERVER or WOLFSSL_WPAS_SMALL)
SSL_OP_NO_RENEGOTIATION	Reject peer-initiated renegotiation
SSL_OP_NO_SESSION_RESUMPTION_ON_RENEGOTIATION	Disable session resumption on renegotiation
SSL_OP_COOKIE_EXCHANGE	Enable HelloVerifyRequest cookie exchange (default-on for DTLS)
SSL_OP_NO_QUERY_MTU	DTLS: do not query the path MTU
SSL_OP_CIPHER_SERVER_PREFERENCE	Server uses its own cipher preference order rather than the client's
SSL_OP_SINGLE_DH_USE	Generate a fresh DH key for every handshake
SSL_OP_SINGLE_ECDH_USE	Generate a fresh ECDH key for every handshake
SSL_OP_EPHEMERAL_RSA	Use ephemeral RSA (legacy, accepted for OpenSSL compatibility)
SSL_OP_DONT_INSERT_EMPTY_FRAGMENTS	Do not insert empty fragments before a record (CBC BEAST workaround)
SSL_OP_PKCS1_CHECK_1 / _2	Accepted for OpenSSL compatibility
SSL_OP_LEGACY_SERVER_CONNECT	Always allow legacy (unsafe) initial connect; defined as 0 — no effect (requires the openssl/ssl.h compatibility header)

Convenience macros and bug-workaround flags (all members of SSL_OP_ALL, accepted for OpenSSL compatibility but otherwise no-ops in wolfSSL):

- SSL_OP_ALL (bitwise OR of all bug-workaround flags below)
- SSL_OP_MICROSOFT_SESS_ID_BUG

- SSL_OP_NETSCAPE_CHALLENGE_BUG
- SSL_OP_NETSCAPE_REUSE_CIPHER_CHANGE_BUG
- SSL_OP_SSLREF2_REUSE_CERT_TYPE_BUG
- SSL_OP_MICROSOFT_BIG_SSLV3_BUFFER
- SSL_OP_MSIE_SSLV2_RSA_PADDING
- SSL_OP_SSLEAY_080_CLIENT_DH_BUG
- SSL_OP_TLS_D5_BUG
- SSL_OP_TLS_BLOCK_PADDING_BUG
- SSL_OP_TLS_ROLLBACK_BUG
- SSL_OP_NETSCAPE_CA_DN_BUG
- SSL_OP_NETSCAPE_DEMO_CIPHER_CHANGE_BUG

Convenience composite:

- SSL_OP_NO_SSL_MASK = SSL_OP_NO_SSLv3 | SSL_OP_NO_TLSv1 | SSL_OP_NO_TLSv1_1 | SSL_OP_NO_TLSv1_2 | SSL_OP_NO_TLSv1_3

Example

```
WOLFSSL_CTX* ctx;
long mask;
mask = wolfSSL_CTX_set_options(ctx,
    SSL_OP_NO_SSLv3 | SSL_OP_NO_TLSv1 | SSL_OP_NO_TLSv1_1 |
    SSL_OP_NO_COMPRESSION | SSL_OP_CIPHER_SERVER_PREFERENCE);
// 'mask' now reflects the accumulated options stored in ctx
```

C.58.4.189 function wolfSSL_set_options

```
long wolfSSL_set_options(
    WOLFSSL * s,
    long op
)
```

This function ORs the bits in op into the options mask of the given WOLFSSL session. The set of recognized "SSL_OP_" / "WOLFSSL_OP_" flags is identical to that documented for wolfSSL_CTX_set_options(); refer to that function for the full options table and the build-option requirements of individual flags. Flags inherited from the parent WOLFSSL_CTX remain set; setting SSL_OP_NO_TLSv1_3 here lowers the session's negotiated minor version to TLS 1.2.

Parameters:

- **s** WOLFSSL session on which to set the options mask.
- **op** the bitmask of SSL_OP_* / WOLFSSL_OP_* flags to OR into the current mask.

See:

- wolfSSL_CTX_set_options
- wolfSSL_clear_options
- wolfSSL_get_options
- wolfSSL_new
- wolfSSL_free

Return: The updated options mask value stored in s on success, or 0 if s is NULL.

Example

```
WOLFSSL* ssl;
long mask;
mask = wolfSSL_set_options(ssl, SSL_OP_NO_TLSv1_3);
// 'mask' now reflects the accumulated options stored in ssl
```

C.58.4.190 function wolfSSL_get_options

```
long wolfSSL_get_options(
    const WOLFSSL * s
)
```

This function returns the current options mask.

Parameters:

- **s** WOLFSSL structure to get options mask from.

See:

- wolfSSL_new
- wolfSSL_free
- wolfSSL_set_options

Return: val Returns the mask value stored in ssl.

Example

```
WOLFSSL* ssl;
unsigned long mask;
mask  = wolfSSL_get_options(ssl);
// check mask
```

C.58.4.191 function wolfSSL_set_tlsext_debug_arg

```
long wolfSSL_set_tlsext_debug_arg(
    WOLFSSL * s,
    void * arg
)
```

This is used to set the debug argument passed around.

Parameters:

- **s** WOLFSSL structure to set argument in.
- **arg** argument to use.

See:

- wolfSSL_new
- wolfSSL_free

Return:

- SSL_SUCCESS On successful setting argument.
- SSL_FAILURE If an NULL ssl passed in.

Example

```
WOLFSSL* ssl;
void* args;
int ret;
// create ssl object
ret  = wolfSSL_set_tlsext_debug_arg(ssl, args);
// check ret value
```

C.58.4.192 function wolfSSL_set_tlsext_debug_callback

```
long wolfSSL_set_tlsext_debug_callback(
    WOLFSSL * s,
    WOLFSSL_TLSEXT_DEBUG_CB cb
)
```

This is used to set the TLS extension debug callback on the object.

Parameters:

- **s** WOLFSSL structure to set the callback in.
- **cb** Callback to invoke for each received TLS extension, or NULL to disable it.

See: wolfSSL_set_tlsext_debug_arg

Return:

- WOLFSSL_SUCCESS On successful setting of the callback.
- WOLFSSL_FAILURE If a NULL ssl is passed in.

The callback (type WOLFSSL_TLSEXT_DEBUG_CB) is invoked once for every TLS extension received during the handshake, in wire order, before the extension is processed. It reports the side of the connection, the extension type, the raw extension content and the argument set with wolfSSL_set_tlsext_debug_arg(). Passing a NULL callback disables it.

Example

```
WOLFSSL* ssl;
long ret;
// create ssl object
ret = wolfSSL_set_tlsext_debug_callback(ssl, my_tlsext_debug_cb);
// check ret value
```

C.58.4.193 function wolfSSL_set_tlsext_status_type

```
long wolfSSL_set_tlsext_status_type(
    WOLFSSL * s,
    int type
)
```

This function is called when the client application request that a server send back an OCSP status response (also known as OCSP stapling).Currently, the only supported type is TLSEXT_STATUSTYPE_ocsp.

Parameters:

- **s** pointer to WOLFSSL struct which is created by SSL_new() function
- **type** ssl extension type which TLSEXT_STATUSTYPE_ocsp is only supported.

See:

- wolfSSL_new
- wolfSSL_CTX_new
- wolfSSL_free
- wolfSSL_CTX_free

Return:

- 1 upon success.
- 0 upon error.
- BAD_FUNC_ARG will be returned when s is NULL.
- SSL_FAILURE will be returned when the type is not OCSP.

Example

```
WOLFSSL *ssl;
WOLFSSL_CTX *ctx;
int ret;
ctx = wolfSSL_CTX_new(wolfSSLv23_server_method());
ssl = wolfSSL_new(ctx);
ret = WolfSSL_set_tlsext_status_type(ssl,TLSEXT_STATUSTYPE_ocsp);
wolfSSL_free(ssl);
wolfSSL_CTX_free(ctx);
```

C.58.4.194 function wolfSSL_get_verify_result

```
long wolfSSL_get_verify_result(
    const WOLFSSL * ssl
)
```

This is used to get the results after trying to verify the peer's certificate.

Parameters:

- **ssl** WOLFSSL structure to get verification results from.

See:

- wolfSSL_new
- wolfSSL_free

Return:

- X509_V_OK On successful verification.
- SSL_FAILURE If an NULL ssl passed in.

Example

```
WOLFSSL* ssl;
long ret;
// attempt/complete handshake
ret  = wolfSSL_get_verify_result(ssl);
// check ret value
```

C.58.4.195 function wolfSSL_ERR_print_errors_fp

```
void wolfSSL_ERR_print_errors_fp(
    XFILE fp,
    int err
)
```

This function converts an error code returned by wolfSSL_get_error() into a more human_readable error string and prints that string to the output file _ fp. err is the error code returned by wolfSSL_get_error() and fp is the file which the error string will be placed in.

Parameters:

- **fp** output file for human-readable error string to be written to.
- **err** error code returned by wolfSSL_get_error().

See:

- wolfSSL_get_error
- wolfSSL_ERR_error_string
- wolfSSL_ERR_error_string_n

- wolfSSL_load_error_strings

Return: none No returns.

Example

```
int err = 0;
WOLFSSL* ssl;
FILE* fp = ...
...
err = wolfSSL_get_error(ssl, 0);
wolfSSL_ERR_print_errors_fp(fp, err);
```

C.58.4.196 function wolfSSL_ERR_print_errors_cb

```
void wolfSSL_ERR_print_errors_cb(
    int(*)(const char *str, size_t len, void *u) cb,
    void * u
)
```

This function uses the provided callback to handle error reporting. The callback function is executed for each error line. The string, length, and userdata are passed into the callback parameters.

Parameters:

- **cb** the callback function.
- **u** userdata to pass into the callback function.

See:

- wolfSSL_get_error
- wolfSSL_ERR_error_string
- wolfSSL_ERR_error_string_n
- wolfSSL_load_error_strings

Return: none No returns.

Example

```
int error_cb(const char *str, size_t len, void *u)
{ fprintf((FILE*)u, "%-*.*s\n", (int)len, (int)len, str); return 0; }
...
FILE* fp = ...
wolfSSL_ERR_print_errors_cb(error_cb, fp);
```

C.58.4.197 function wolfSSL_CTX_set_psk_client_callback

```
void wolfSSL_CTX_set_psk_client_callback(
    WOLFSSL_CTX * ctx,
    wc_psk_client_callback cb
)
```

The function sets the client_psk_cb member of the WOLFSSL_CTX structure.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **cb** wc_psk_client_callback is a function pointer that will be stored in the WOLFSSL_CTX structure. Return value is the key length on success or zero on error. unsigned int *(wc_psk_client_callback) PSK client callback parameters: WOLFSSL* ssl - Pointer to the wolfSSL structure const char* hint - A stored string that could be displayed to provide a hint to the user. char* identity - The ID will be

stored here. unsigned int id_max_len - Size of the ID buffer. unsigned char* key - The key will be stored here. unsigned int key_max_len - The max size of the key.

See:

- wolfSSL_set_psk_client_callback
- wolfSSL_set_psk_server_callback
- wolfSSL_CTX_set_psk_server_callback
- wolfSSL_CTX_set_psk_client_callback

Return: none No returns.

Example

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new( protocol def );
…
static WC_INLINE unsigned int my_psk_client_cb(WOLFSSL* ssl, const char* hint,
char* identity, unsigned int id_max_len, unsigned char* key,
Unsigned int key_max_len){
…
wolfSSL_CTX_set_psk_client_callback(ctx, my_psk_client_cb);
```

C.58.4.198 function wolfSSL_set_psk_client_callback

```
void wolfSSL_set_psk_client_callback(
    WOLFSSL * ssl,
    wc_psk_client_callback cb
)
```

Sets the PSK client side callback.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **cb** a function pointer to type wc_psk_client_callback. Return value is the key length on success or zero on error. unsigned int (*wc_psk_client_callback) PSK client callback parameters: WOLFSSL ssl - Pointer to the wolfSSL structure const char* hint - A stored string that could be displayed to provide a hint to the user. char* identity - The ID will be stored here. unsigned int id_max_len - Size of the ID buffer. unsigned char* key - The key will be stored here. unsigned int key_max_len - The max size of the key.

See:

- wolfSSL_CTX_set_psk_client_callback
- wolfSSL_CTX_set_psk_server_callback
- wolfSSL_set_psk_server_callback

Return: none No returns.

Example

```
WOLFSSL* ssl;
static WC_INLINE unsigned int my_psk_client_cb(WOLFSSL* ssl, const char* hint,
char* identity, unsigned int id_max_len, unsigned char* key,
Unsigned int key_max_len){
…
if(ssl){
wolfSSL_set_psk_client_callback(ssl, my_psk_client_cb);
} else {
    // could not set callback
}
```

C.58.4.199 function wolfSSL_get_psk_identity_hint

```
const char * wolfSSL_get_psk_identity_hint(
    const WOLFSSL *
)
```

This function returns the psk identity hint.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_get_psk_identity

Return:

- pointer a const char pointer to the value that was stored in the arrays member of the WOLFSSL structure is returned.
- NULL returned if the WOLFSSL or Arrays structures are NULL.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
char* idHint;
...
idHint = wolfSSL_get_psk_identity_hint(ssl);
if(idHint){
    // The hint was retrieved
    return idHint;
} else {
    // Hint wasn't successfully retrieved
}
```

C.58.4.200 function wolfSSL_get_psk_identity

```
const char * wolfSSL_get_psk_identity(
    const WOLFSSL *
)
```

The function returns a constant pointer to the client_identity member of the Arrays structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_get_psk_identity_hint
- wolfSSL_use_psk_identity_hint

Return:

- string the string value of the client_identity member of the Arrays structure.
- NULL if the WOLFSSL structure is NULL or if the Arrays member of the WOLFSSL structure is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
const char* pskID;
...
pskID = wolfSSL_get_psk_identity(ssl);
```

```
if(pskID == NULL){
    // There is not a value in pskID
}
```

C.58.4.201 function wolfSSL_CTX_use_psk_identity_hint

```
int wolfSSL_CTX_use_psk_identity_hint(
    WOLFSSL_CTX * ctx,
    const char * hint
)
```

This function stores the hint argument in the server_hint member of the WOLFSSL_CTX structure.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **hint** a constant char pointer that will be copied to the WOLFSSL_CTX structure.

See: wolfSSL_use_psk_identity_hint

Return: SSL_SUCCESS returned for successful execution of the function.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
const char* hint;
int ret;
…
ret = wolfSSL_CTX_use_psk_identity_hint(ctx, hint);
if(ret == SSL_SUCCESS){
    // Function was successful.
return ret;
} else {
    // Failure case.
}
```

C.58.4.202 function wolfSSL_use_psk_identity_hint

```
int wolfSSL_use_psk_identity_hint(
    WOLFSSL * ssl,
    const char * hint
)
```

This function stores the hint argument in the server_hint member of the Arrays structure within the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **hint** a constant character pointer that holds the hint to be saved in memory.

See: wolfSSL_CTX_use_psk_identity_hint

Return:

- SSL_SUCCESS returned if the hint was successfully stored in the WOLFSSL structure.
- SSL_FAILURE returned if the WOLFSSL or Arrays structures are NULL.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
const char* hint;
...
if(wolfSSL_use_psk_identity_hint(ssl, hint) != SSL_SUCCESS){
    // Handle failure case.
}
```

C.58.4.203 function wolfSSL_CTX_set_psk_server_callback

```
void wolfSSL_CTX_set_psk_server_callback(
    WOLFSSL_CTX * ctx,
    wc_psk_server_callback cb
)
```

This function sets the psk callback for the server side in the WOLFSSL_CTX structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **cb** a function pointer for the callback and will be stored in the WOLFSSL_CTX structure. Return value is the key length on success or zero on error. unsigned int *(wc_psk_server_callback) PSK server callback parameters WOLFSSL* ssl - Pointer to the wolfSSL structure char* identity - The ID will be stored here. unsigned char* key - The key will be stored here. unsigned int key_max_len - The max size of the key.

See:

- wc_psk_server_callback
- wolfSSL_set_psk_client_callback
- wolfSSL_set_psk_server_callback
- wolfSSL_CTX_set_psk_client_callback

Return: none No returns.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
…
static unsigned int my_psk_server_cb(WOLFSSL* ssl, const char* identity,
                        unsigned char* key, unsigned int key_max_len)
{
    // Function body.
}
…
if(ctx != NULL){
    wolfSSL_CTX_set_psk_server_callback(ctx, my_psk_server_cb);
} else {
    // The CTX object was not properly initialized.
}
```

C.58.4.204 function wolfSSL_set_psk_server_callback

```
void wolfSSL_set_psk_server_callback(
    WOLFSSL * ssl,
    wc_psk_server_callback cb
)
```

Sets the psk callback for the server side by setting the WOLFSSL structure options members.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **cb** a function pointer for the callback and will be stored in the WOLFSSL structure. Return value is the key length on success or zero on error. unsigned int (*wc_psk_server_callback) PSK server callback parameters WOLFSSL* ssl - Pointer to the wolfSSL structure char* identity - The ID will be stored here. unsigned char* key - The key will be stored here. unsigned int key_max_len - The max size of the key.

See:

- wolfSSL_set_psk_client_callback
- wolfSSL_CTX_set_psk_server_callback
- wolfSSL_CTX_set_psk_client_callback
- wolfSSL_get_psk_identity_hint
- wc_psk_server_callback
- InitSuites

Return: none No returns.

Example

```
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
…
static unsigned int my_psk_server_cb(WOLFSSL* ssl, const char* identity,
                        unsigned char* key, unsigned int key_max_len)
{
    // Function body.
}
…
if(ssl != NULL && cb != NULL){
    wolfSSL_set_psk_server_callback(ssl, my_psk_server_cb);
}
```

C.58.4.205 function wolfSSL_set_psk_callback_ctx

```
int wolfSSL_set_psk_callback_ctx(
    WOLFSSL * ssl,
    void * psk_ctx
)
```

Sets a PSK user context in the WOLFSSL structure options member.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **psk_ctx** void pointer to user PSK context

See:

- wolfSSL_get_psk_callback_ctx
- wolfSSL_CTX_set_psk_callback_ctx
- wolfSSL_CTX_get_psk_callback_ctx

Return: WOLFSSL_SUCCESS or WOLFSSL_FAILURE

C.58.4.206 function wolfSSL_CTX_set_psk_callback_ctx

```
int wolfSSL_CTX_set_psk_callback_ctx(
    WOLFSSL_CTX * ctx,
    void * psk_ctx
)
```

Sets a PSK user context in the WOLFSSL_CTX structure.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **psk_ctx** void pointer to user PSK context

See:

- wolfSSL_set_psk_callback_ctx
- wolfSSL_get_psk_callback_ctx
- wolfSSL_CTX_get_psk_callback_ctx

Return: WOLFSSL_SUCCESS or WOLFSSL_FAILURE

C.58.4.207 function wolfSSL_get_psk_callback_ctx

```
void * wolfSSL_get_psk_callback_ctx(
    WOLFSSL * ssl
)
```

Get a PSK user context in the WOLFSSL structure options member.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_set_psk_callback_ctx
- wolfSSL_CTX_set_psk_callback_ctx
- wolfSSL_CTX_get_psk_callback_ctx

Return: void pointer to user PSK context

C.58.4.208 function wolfSSL_CTX_get_psk_callback_ctx

```
void * wolfSSL_CTX_get_psk_callback_ctx(
    WOLFSSL_CTX * ctx
)
```

Get a PSK user context in the WOLFSSL_CTX structure.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

See:

- wolfSSL_CTX_set_psk_callback_ctx
- wolfSSL_set_psk_callback_ctx
- wolfSSL_get_psk_callback_ctx

Return: void pointer to user PSK context

C.58.4.209 function wolfSSL_CTX_allow_anon_cipher

```
int wolfSSL_CTX_allow_anon_cipher(
    WOLFSSL_CTX * ctx
)
```

This function enables the havAnon member of the CTX structure if HAVE_ANON is defined during compilation.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

See: none

Return:

- SSL_SUCCESS returned if the function executed successfully and the haveAnnon member of the CTX is set to 1.
- SSL_FAILURE returned if the CTX structure was NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
#ifdef HAVE_ANON
if(cipherList == NULL){
    wolfSSL_CTX_allow_anon_cipher(ctx);
    if(wolfSSL_CTX_set_cipher_list(ctx, “ADH_AES128_SHA”) != SSL_SUCCESS){
        // failure case
    }
}
#endif
```

C.58.4.210 function wolfSSLv23_server_method

```
WOLFSSL_METHOD * wolfSSLv23_server_method(
    void
)
```

The wolfSSLv23_server_method() function is used to indicate that the application is a server and will support clients connecting with protocol version from SSL 3.0 _ TLS 1.3. This function allocates memory for and initializes a new WOLFSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Parameters:

- **none** No parameters

See:

- wolfSSLv3_server_method
- wolfTLSv1_server_method
- wolfTLSv1_1_server_method
- wolfTLSv1_2_server_method
- wolfTLSv1_3_server_method
- wolfDTLSv1_server_method
- wolfSSL_CTX_new

Return:

- pointer If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- Failure If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfSSLv23_server_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.58.4.211 function wolfSSL_state

```
int wolfSSL_state(
    WOLFSSL * ssl
)
```

This is used to get the internal error state of the WOLFSSL structure.

See:

- wolfSSL_new
- wolfSSL_free

Return:

- wolfssl_error returns ssl error state, usually a negative
- BAD_FUNC_ARG if ssl is NULL.
- ssl WOLFSSL structure to get state from.

Example

```
WOLFSSL* ssl;
int ret;
// create ssl object
ret  = wolfSSL_state(ssl);
// check ret value
```

C.58.4.212 function wolfSSL_get_peer_certificate

```
WOLFSSL_X509 * wolfSSL_get_peer_certificate(
    WOLFSSL * ssl
)
```

This function gets the peer's certificate.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_subject_name

- wolfSSL_X509_get_isCA

Return:

- pointer a pointer to the peerCert member of the WOLFSSL_X509 structure if it exists.
- 0 returned if the peer certificate issuer size is not defined.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
WOLFSSL_X509* peerCert = wolfSSL_get_peer_certificate(ssl);

if(peerCert){
    // You have a pointer peerCert to the peer certification
}
```

C.58.4.213 function wolfSSL_want_read

```
int wolfSSL_want_read(
    WOLFSSL * ssl
)
```

This function is similar to calling wolfSSL_get_error() and getting SSL_ERROR_WANT_READ in return. If the underlying error state is SSL_ERROR_WANT_READ, this function will return 1, otherwise, 0.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See:

- wolfSSL_want_write
- wolfSSL_get_error

Return:

- 1 wolfSSL_get_error() would return SSL_ERROR_WANT_READ, the underlying I/O has data available for reading.
- 0 There is no SSL_ERROR_WANT_READ error state.

Example

```
int ret;
WOLFSSL* ssl = 0;
...

ret = wolfSSL_want_read(ssl);
if (ret == 1) {
    // underlying I/O has data available for reading (SSL_ERROR_WANT_READ)
}
```

C.58.4.214 function wolfSSL_want_write

```
int wolfSSL_want_write(
    WOLFSSL * ssl
)
```

This function is similar to calling wolfSSL_get_error() and getting SSL_ERROR_WANT_WRITE in return. If the underlying error state is SSL_ERROR_WANT_WRITE, this function will return 1, otherwise, 0.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See:

- wolfSSL_want_read
- wolfSSL_get_error

Return:

- 1 wolfSSL_get_error() would return SSL_ERROR_WANT_WRITE, the underlying I/O needs data to be written in order for progress to be made in the underlying SSL connection.
- 0 There is no SSL_ERROR_WANT_WRITE error state.

Example

```
int ret;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_want_write(ssl);
if (ret == 1) {
    // underlying I/O needs data to be written (SSL_ERROR_WANT_WRITE)
}
```

C.58.4.215 function wolfSSL_want

```
int wolfSSL_want(
    WOLFSSL * ssl
)
```

This function reports which I/O operation, if any, the SSL session is waiting on. It reflects the same state that wolfSSL_want_read() and wolfSSL_want_write() report individually. Unlike those two, which are always available, this function is only built when OPENSSL_EXTRA is defined.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See:

- wolfSSL_want_read
- wolfSSL_want_write
- wolfSSL_get_error

Return:

- WOLFSSL_READING when the underlying I/O needs data to be read before progress can be made.
- WOLFSSL_WRITING when the underlying I/O needs data to be written before progress can be made.
- WOLFSSL_NOTHING when the session is not waiting on the underlying I/O, or ssl is NULL.

Example

```
WOLFSSL* ssl = 0;
...

if (wolfSSL_want(ssl) == WOLFSSL_READING) {
    // wait for the socket to become readable, then retry
}
```

C.58.4.216 function wolfSSL_check_domain_name

```
int wolfSSL_check_domain_name(
    WOLFSSL * ssl,
    const char * dn
)
```

wolfSSL by default checks the peer certificate for a valid date range and a verified signature. Calling this function before wolfSSL_connect() or wolfSSL_accept() will add a domain name check to the list of checks to perform. dn holds the domain name to check against the peer certificate when it's received.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **dn** domain name to check against the peer certificate when received.

See: none

Return:

- SSL_SUCCESS upon success.
- SSL_FAILURE will be returned if a memory error was encountered.

Example

```
int ret = 0;
WOLFSSL* ssl;
char* domain = (char*) "www.yassl.com";
...

ret = wolfSSL_check_domain_name(ssl, domain);
if (ret != SSL_SUCCESS) {
   // failed to enable domain name check
}
```

C.58.4.217 function wolfSSL_check_ip_address

```
int wolfSSL_check_ip_address(
    WOLFSSL * ssl,
    const char * ipaddr
)
```

Calling this function before wolfSSL_connect() or wolfSSL_accept() adds an IP-address identity check against the peer certificate SAN iPAddress entries.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **ipaddr** NULL-terminated ASCII IP address string to verify against the peer certificate.

See: wolfSSL_check_domain_name

Return:

- SSL_SUCCESS upon success.
- SSL_FAILURE if parameters are invalid or memory allocation fails.

Example

```
int ret = 0;
WOLFSSL* ssl;
const char* ip = "127.0.0.1";
...
```

```
ret = wolfSSL_check_ip_address(ssl, ip);
if (ret != SSL_SUCCESS) {
   // failed to enable IP check
}
```

C.58.4.218 function wolfSSL_Init

```
int wolfSSL_Init(
    void
)
```

Initializes the wolfSSL library for use. Must be called once per application and before any other call to the library.

See: wolfSSL_Cleanup

Return:

- SSL_SUCCESS If successful the call will return.
- BAD_MUTEX_E is an error that may be returned.
- WC_INIT_E wolfCrypt initialization error returned.

Example

```
int ret = 0;
ret = wolfSSL_Init();
if (ret != SSL_SUCCESS) {
    failed to initialize wolfSSL library
}
```

C.58.4.219 function wolfSSL_Cleanup

```
int wolfSSL_Cleanup(
    void
)
```

Un-initializes the wolfSSL library from further use. Doesn't have to be called, though it will free any resources used by the library.

See: wolfSSL_Init

Return:

- SSL_SUCCESS return no errors.
- BAD_MUTEX_E a mutex error return.]

Example

```
wolfSSL_Cleanup();
```

C.58.4.220 function wolfSSL_lib_version

```
const char * wolfSSL_lib_version(
    void
)
```

This function returns the current library version.

Parameters:

- **none** No parameters.

See: word32_wolfSSL_lib_version_hex

Return: LIBWOLFSSL_VERSION_STRING a const char pointer defining the version.

Example

```
char version[MAXSIZE];
version = wolfSSL_KeepArrays();
…
if(version != ExpectedVersion){
    // Handle the mismatch case
}
```

C.58.4.221 function wolfSSL_lib_version_hex

```
word32 wolfSSL_lib_version_hex(
    void
)
```

This function returns the current library version in hexadecimal notation.

Parameters:

- **none** No parameters.

See: wolfSSL_lib_version

Return: LILBWOLFSSL_VERSION_HEX returns the hexadecimal version defined in wolfssl/version.h.

Example

```
word32 libV;
libV = wolfSSL_lib_version_hex();

if(libV != EXPECTED_HEX){
    // How to handle an unexpected value
} else {
    // The expected result for libV
}
```

C.58.4.222 function wolfSSL_negotiate

```
int wolfSSL_negotiate(
    WOLFSSL * ssl
)
```

Performs the actual connect or accept based on the side of the SSL method. If called from the client side then an wolfSSL_connect() is done while a wolfSSL_accept() is performed if called from the server side.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See:

- SSL_connect
- SSL_accept

Return:

- SSL_SUCCESS will be returned if successful. (Note, older versions will return 0.)

- SSL_FATAL_ERROR will be returned if the underlying call resulted in an error. Use wolfSSL_get_error() to get a specific error code.

If decrypted application data is waiting to be read, the handshake is not continued and wolfSSL_get_error() returns APP_DATA_READY. Read the data with wolfSSL_read() and then call wolfSSL_negotiate() again.

Example

```
int ret = SSL_FATAL_ERROR;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_negotiate(ssl);
if (ret == SSL_FATAL_ERROR) {
    // SSL establishment failed
int error_code = wolfSSL_get_error(ssl);
...
}
...
```

C.58.4.223 function wolfSSL_set_compression

```
int wolfSSL_set_compression(
    WOLFSSL * ssl
)
```

Turns on the ability to use compression for the SSL connection. Both sides must have compression turned on otherwise compression will not be used. The zlib library performs the actual data compression. To compile into the library use –with-libz for the configure system and define HAVE_LIBZ otherwise. Keep in mind that while compressing data before sending decreases the actual size of the messages being sent and received, the amount of data saved by compression usually takes longer in time to analyze than it does to send it raw on all but the slowest of networks.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See: none

Return:

- SSL_SUCCESS upon success.
- NOT_COMPILED_IN will be returned if compression support wasn't built into the library.
- BAD_FUNC_ARG will be returned if ssl is NULL or is a DTLS session.

Record layer compression was removed in TLS 1.3 (RFC 8446 section 5.2). A connection that negotiates TLS 1.3 or DTLS 1.3 therefore completes uncompressed and this request is silently dropped; the call still returns SSL_SUCCESS, so check the negotiated protocol version rather than this return value to learn whether compression is actually in use. For the same reason a ClientHello that offers TLS 1.3 never advertises zlib.

Compression is not available over DTLS at all. zlib keeps one deflate stream running across records, so a datagram that is lost, duplicated or reordered would desync the peer for the rest of the connection. Since the transport is known when the WOLFSSL object is created, this is reported rather than dropped.

Example

```
int ret = 0;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_set_compression(ssl);
```

```
if (ret == SSL_SUCCESS) {
    // successfully enabled compression for SSL session
}
```

C.58.4.224 function wolfSSL_set_timeout

```
int wolfSSL_set_timeout(
    WOLFSSL * ssl,
    unsigned int to
)
```

This function sets the SSL session timeout value in seconds.

Parameters:

- **ssl** pointer to the SSL object, created with wolfSSL_new().
- **to** value, in seconds, used to set the SSL session timeout.

See:

- wolfSSL_get1_session
- wolfSSL_set_session

Return:

- SSL_SUCCESS will be returned upon successfully setting the session.
- BAD_FUNC_ARG will be returned if ssl is NULL.

Example

```
int ret = 0;
WOLFSSL* ssl = 0;
...

ret = wolfSSL_set_timeout(ssl, 500);
if (ret != SSL_SUCCESS) {
    // failed to set session timeout value
}
...
```

C.58.4.225 function wolfSSL_CTX_set_timeout

```
int wolfSSL_CTX_set_timeout(
    WOLFSSL_CTX * ctx,
    unsigned int to
)
```

This function sets the timeout value for SSL sessions, in seconds, for the specified SSL context.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **to** session timeout value in seconds.

See:

- wolfSSL_flush_sessions
- wolfSSL_get1_session
- wolfSSL_set_session
- wolfSSL_get_sessionID
- wolfSSL_CTX_set_session_cache_mode

Return:

- the previous timeout value, if WOLFSSL_ERROR_CODE_OPENSSL is
- defined on success. If not defined, SSL_SUCCESS will be returned.
- BAD_FUNC_ARG will be returned when the input context (ctx) is null.

Example

```
WOLFSSL_CTX*    ctx    = 0;
...
ret = wolfSSL_CTX_set_timeout(ctx, 500);
if (ret != SSL_SUCCESS) {
    // failed to set session timeout value
}
```

C.58.4.226 function wolfSSL_get_peer_chain

```
WOLFSSL_X509_CHAIN * wolfSSL_get_peer_chain(
    WOLFSSL * ssl
)
```

Retrieves the peer's certificate chain.

Parameters:

- **ssl** pointer to a valid WOLFSSL structure.

See:

- wolfSSL_get_chain_count
- wolfSSL_get_chain_length
- wolfSSL_get_chain_cert
- wolfSSL_get_chain_cert_pem

Return:

- chain If successful the call will return the peer's certificate chain.
- 0 will be returned if an invalid WOLFSSL pointer is passed to the function.

Example

none

C.58.4.227 function wolfSSL_get_chain_count

```
int wolfSSL_get_chain_count(
    WOLFSSL_X509_CHAIN * chain
)
```

Retrieve's the peers certificate chain count.

Parameters:

- **chain** pointer to a valid WOLFSSL_X509_CHAIN structure.

See:

- wolfSSL_get_peer_chain
- wolfSSL_get_chain_length
- wolfSSL_get_chain_cert
- wolfSSL_get_chain_cert_pem

Return:

- Success If successful the call will return the peer's certificate chain count.
- 0 will be returned if an invalid chain pointer is passed to the function.

Example

none

C.58.4.228 function wolfSSL_get_chain_length

```
int wolfSSL_get_chain_length(
    WOLFSSL_X509_CHAIN * chain,
    int idx
)
```

Retrieves the peer's ASN1.DER certificate length in bytes at index (idx).

Parameters:

- **chain** pointer to a valid WOLFSSL_X509_CHAIN structure.
- **idx** index to start of chain.

See:

- wolfSSL_get_peer_chain
- wolfSSL_get_chain_count
- wolfSSL_get_chain_cert
- wolfSSL_get_chain_cert_pem

Return:

- Success If successful the call will return the peer's certificate length in bytes by index.
- 0 will be returned if an invalid chain pointer is passed to the function, or if the slot at idx holds no certificate because the one presented there was too large to store.

Example

none

C.58.4.229 function wolfSSL_get_chain_cert

```
unsigned char * wolfSSL_get_chain_cert(
    WOLFSSL_X509_CHAIN * chain,
    int idx
)
```

Retrieves the peer's ASN1.DER certificate at index (idx).

Parameters:

- **chain** pointer to a valid WOLFSSL_X509_CHAIN structure.
- **idx** index to start of chain.

See:

- wolfSSL_get_peer_chain
- wolfSSL_get_chain_count
- wolfSSL_get_chain_length
- wolfSSL_get_chain_cert_pem

Return:

- Success If successful the call will return the peer's certificate by index.

- 0 will be returned if an invalid chain pointer is passed to the function, or if the slot at idx holds no certificate because the one presented there was too large to store. Indices inside wolfSSL_get_chain_count() can be empty, so check wolfSSL_get_chain_length() before using the result.

Example

none

C.58.4.230 function wolfSSL_get_chain_X509

```
WOLFSSL_X509 * wolfSSL_get_chain_X509(
    WOLFSSL_X509_CHAIN * chain,
    int idx
)
```

This function gets the peer's wolfSSL_X509_certificate at index (idx) from the chain of certificates.

Parameters:

- **chain** a pointer to the WOLFSSL_X509_CHAIN used for no dynamic memory SESSION_CACHE.
- **idx** the index of the WOLFSSL_X509 certificate.

See:

- InitDecodedCert
- ParseCertRelative
- CopyDecodedToX509

Return: pointer returns a pointer to a WOLFSSL_X509 structure.

Note that it is the user's responsibility to free the returned memory by calling wolfSSL_FreeX509(). NULL is also returned when the slot at idx holds no certificate because the one presented there was too large to store, which can happen for an index inside wolfSSL_get_chain_count().

Example

```
WOLFSSL_X509_CHAIN* chain = &session->chain;
int idx = 999; // set idx
...
WOLFSSL_X509_CHAIN ptr;
prt = wolfSSL_get_chain_X509(chain, idx);

if(ptr != NULL){
    // ptr contains the cert at the index specified
    wolfSSL_FreeX509(ptr);
} else {
    // ptr is NULL
}
```

C.58.4.231 function wolfSSL_get_chain_cert_pem

```
int wolfSSL_get_chain_cert_pem(
    WOLFSSL_X509_CHAIN * chain,
    int idx,
    unsigned char * buf,
    int inLen,
    int * outLen
)
```

Retrieves the peer's PEM certificate at index (idx).

Parameters:

- **chain** pointer to a valid WOLFSSL_X509_CHAIN structure.
- **idx** index to start of chain.
- **buf** buffer to hold the PEM certificate. May be NULL to ask for the required length only.
- **inLen** length of buf in bytes.
- **outLen** length of the PEM data in bytes.

See:

- wolfSSL_get_peer_chain
- wolfSSL_get_chain_count
- wolfSSL_get_chain_length
- wolfSSL_get_chain_cert

Return:

- SSL_SUCCESS will be returned on success.
- SSL_FAILURE will be returned when the certificate cannot be converted.
- BAD_FUNC_ARG will be returned when chain is NULL, idx is out of range, outLen is NULL, or inLen is negative. An idx inside wolfSSL_get_chain_count() whose slot holds no certificate, because the one presented there was too large to store, is reported the same way.
- BUFFER_E will be returned when inLen is too small to hold the PEM output, in a build that converts with wc_DerToPem(). A build that has only WOLFSSL_PEM_TO_DER reports that case as BAD_FUNC_ARG, so a caller growing its buffer on a short write should accept either.
- LENGTH_ONLY_E will be returned when buf is NULL, with the required length returned through outLen.

Example

none

C.58.4.232 function wolfSSL_get_sessionID

```
const unsigned char * wolfSSL_get_sessionID(
    const WOLFSSL_SESSION * s
)
```

Retrieves the session's ID. The session ID is always 32 bytes long.

Parameters:

- **session** pointer to a valid wolfssl session.

See: SSL_get_session

Return: id The session ID.

Example

none

C.58.4.233 function wolfSSL_X509_get_serial_number

```
int wolfSSL_X509_get_serial_number(
    WOLFSSL_X509 * x509,
    unsigned char * in,
    int * inOutSz
)
```

Retrieves the peer's certificate serial number. The serial number buffer (in) should be at least 32 bytes long and be provided as the *inOutSz argument as input. After calling the function* inOutSz will hold the actual length in bytes written to the in buffer.

Parameters:

- **in** The serial number buffer and should be at least 32 bytes long
- **inOutSz** will hold the actual length in bytes written to the in buffer.

See: SSL_get_peer_certificate

Return:

- SSL_SUCCESS upon success.
- BAD_FUNC_ARG will be returned if a bad function argument was encountered.

Example

none

C.58.4.234 function wolfSSL_X509_get_subjectCN

```
char * wolfSSL_X509_get_subjectCN(
    WOLFSSL_X509 *
)
```

Returns the common name of the subject from the certificate.

Parameters:

- **x509** a pointer to a WOLFSSL_X509 structure containing certificate information.

See:

- wolfSSL_X509_Name_get_entry
- wolfSSL_X509_get_next_altname
- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_subject_name

Return:

- NULL returned if the x509 structure is null
- string a string representation of the subject's common name is returned upon success

Example

```
WOLFSSL_X509 x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                        DYNAMIC_TYPE_X509);
...
int x509Cn = wolfSSL_X509_get_subjectCN(x509);
if(x509Cn == NULL){
    // Deal with NULL case
} else {
    // x509Cn contains the common name
}
```

C.58.4.235 function wolfSSL_X509_get_der

```
const unsigned char * wolfSSL_X509_get_der(
    WOLFSSL_X509 * x509,
    int * outSz
)
```

This function gets the DER encoded certificate in the WOLFSSL_X509 struct.

Parameters:

- **x509** a pointer to a WOLFSSL_X509 structure containing certificate information.
- **outSz** length of the derBuffer member of the WOLFSSL_X509 struct.

See:

- wolfSSL_X509_version
- wolfSSL_X509_Name_get_entry
- wolfSSL_X509_get_next_altname
- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_subject_name

Return:

- buffer This function returns the DerBuffer structure's buffer member, which is of type byte.
- NULL returned if the x509 or outSz parameter is NULL.

Example

```
WOLFSSL_X509 x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                        DYNAMIC_TYPE_X509);
int* outSz; // initialize
...
byte* x509Der = wolfSSL_X509_get_der(x509, outSz);
if(x509Der == NULL){
    // Failure case one of the parameters was NULL
}
```

C.58.4.236 function wolfSSL_X509_get_notAfter

```
WOLFSSL_ASN1_TIME * wolfSSL_X509_get_notAfter(
    WOLFSSL_X509 *
)
```

This function checks to see if x509 is NULL and if it's not, it returns the notAfter member of the x509 struct.

Parameters:

- **x509** a pointer to the WOLFSSL_X509 struct.

See: wolfSSL_X509_get_notBefore

Return:

- pointer to struct with ASN1_TIME to the notAfter member of the x509 struct.
- NULL returned if the x509 object is NULL.

Example

```
WOLFSSL_X509* x509 = (WOLFSSL_X509)XMALOC(sizeof(WOLFSSL_X509), NULL,
DYNAMIC_TYPE_X509) ;
...
const WOLFSSL_ASN1_TIME* notAfter = wolfSSL_X509_get_notAfter(x509);
if(notAfter == NULL){
    // Failure case, the x509 object is null.
}
```

C.58.4.237 function wolfSSL_X509_version

```
int wolfSSL_X509_version(
    WOLFSSL_X509 * x509
)
```

This function retrieves the version of the X509 certificate.

Parameters:

- **x509** a pointer to a WOLFSSL_X509 structure.

See:

- wolfSSL_X509_get_subject_name
- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_isCA
- wolfSSL_get_peer_certificate

Return:

- 0 returned if the x509 structure is NULL.
- version the version stored in the x509 structure will be returned.

Example

```
WOLFSSL_X509* x509;
int version;
...
version = wolfSSL_X509_version(x509);
if(!version){
    // The function returned 0, failure case.
}
```

C.58.4.238 function wolfSSL_X509_d2i_fp

```
WOLFSSL_X509 * wolfSSL_X509_d2i_fp(
    WOLFSSL_X509 ** x509,
    FILE * file
)
```

If NO_STDIO_FILESYSTEM is defined this function will allocate heap memory, initialize a WOLFSSL_X509 structure and return a pointer to it.

Parameters:

- **x509** a pointer to a WOLFSSL_X509 pointer.
- **file** a defined type that is a pointer to a FILE.

See:

- wolfSSL_X509_d2i
- XFTELL
- XREWIND
- XFSEEK

Return:

- *WOLFSSL_X509 WOLFSSL_X509 structure pointer is returned if the function executes successfully.
- NULL if the call to XFTELL macro returns a negative value.

Example

```
WOLFSSL_X509* x509a = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
DYNAMIC_TYPE_X509);
WOLFSSL_X509** x509 = x509a;
XFILE file;  (mapped to struct fs_file*)
...
WOLFSSL_X509* newX509 = wolfSSL_X509_d2i_fp(x509, file);
if(newX509 == NULL){
    // The function returned NULL
}
```

C.58.4.239 function wolfSSL_X509_load_certificate_file

```
WOLFSSL_X509 * wolfSSL_X509_load_certificate_file(
    const char * fname,
    int format
)
```

The function loads the x509 certificate into memory.

Parameters:

- **fname** the certificate file to be loaded.
- **format** the format of the certificate.

See:

- InitDecodedCert
- PemToDer
- wolfSSL_get_certificate
- AssertNotNull

Return:

- pointer a successful execution returns pointer to a WOLFSSL_X509 structure.
- NULL returned if the certificate was not able to be written.

Example

```
#define cliCert    "certs/client-cert.pem"
…
X509* x509;
…
x509 = wolfSSL_X509_load_certificate_file(cliCert, SSL_FILETYPE_PEM);
AssertNotNull(x509);
```

C.58.4.240 function wolfSSL_X509_get_device_type

```
unsigned char * wolfSSL_X509_get_device_type(
    WOLFSSL_X509 * x509,
    unsigned char * in,
    int * inOutSz
)
```

This function copies the device type from the x509 structure to the buffer.

Parameters:

- **x509** pointer to a WOLFSSL_X509 structure, created with WOLFSSL_X509_new().
- **in** a pointer to a byte type that will hold the device type (the buffer).

- **inOutSz** the minimum of either the parameter inOutSz or the deviceTypeSz member of the x509 structure.

See:

- wolfSSL_X509_get_hw_type
- wolfSSL_X509_get_hw_serial_number
- wolfSSL_X509_d2i

Return:

- pointer returns a byte pointer holding the device type from the x509 structure.
- NULL returned if the buffer size is NULL.

Example

```
WOLFSSL_X509* x509 = (WOLFSSL_X509)XMALOC(sizeof(WOLFSSL_X509), NULL,
DYNAMIC_TYPE_X509);
byte* in;
int* inOutSz;
...
byte* deviceType = wolfSSL_X509_get_device_type(x509, in, inOutSz);

if(!deviceType){
    // Failure case, NULL was returned.
}
```

C.58.4.241 function wolfSSL_X509_get_hw_type

```
unsigned char * wolfSSL_X509_get_hw_type(
    WOLFSSL_X509 * x509,
    unsigned char * in,
    int * inOutSz
)
```

The function copies the hwType member of the WOLFSSL_X509 structure to the buffer.

Parameters:

- **x509** a pointer to a WOLFSSL_X509 structure containing certificate information.
- **in** pointer to type byte that represents the buffer.
- **inOutSz** pointer to type int that represents the size of the buffer.

See:

- wolfSSL_X509_get_hw_serial_number
- wolfSSL_X509_get_device_type

Return:

- byte The function returns a byte type of the data previously held in the hwType member of the WOLFSSL_X509 structure.
- NULL returned if inOutSz is NULL.

Example

```
WOLFSSL_X509* x509;  // X509 certificate
byte* in;  // initialize the buffer
int* inOutSz;  // holds the size of the buffer
...
byte* hwType = wolfSSL_X509_get_hw_type(x509, in, inOutSz);
```

```
if(hwType == NULL){
    // Failure case function returned NULL.
}
```

C.58.4.242 function wolfSSL_X509_get_hw_serial_number

```
unsigned char * wolfSSL_X509_get_hw_serial_number(
    WOLFSSL_X509 * x509,
    unsigned char * in,
    int * inOutSz
)
```

This function returns the hwSerialNum member of the x509 object.

Parameters:

- **x509** pointer to a WOLFSSL_X509 structure containing certificate information.
- **in** a pointer to the buffer that will be copied to.
- **inOutSz** a pointer to the size of the buffer.

See:

- wolfSSL_X509_get_subject_name
- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_isCA
- wolfSSL_get_peer_certificate
- wolfSSL_X509_version

Return: pointer the function returns a byte pointer to the in buffer that will contain the serial number loaded from the x509 object.

Example

```
char* serial;
byte* in;
int* inOutSz;
WOLFSSL_X509 x509;
...
serial = wolfSSL_X509_get_hw_serial_number(x509, in, inOutSz);

if(serial == NULL || serial <= 0){
    // Failure case
}
```

C.58.4.243 function wolfSSL_connect_cert

```
int wolfSSL_connect_cert(
    WOLFSSL * ssl
)
```

This function is called on the client side and initiates an SSL/TLS handshake with a server only long enough to get the peer's certificate chain. When this function is called, the underlying communication channel has already been set up. wolfSSL_connect_cert() works with both blocking and non_blocking I/O. When the underlying I/O is non_blocking, wolfSSL_connect_cert() will return when the underlying I/O could not satisfy the needs of wolfSSL_connect_cert() to continue the handshake. In this case, a call to wolfSSL_get_error() will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_connect_cert() when the underlying I/O is ready and wolfSSL will pick up where it left off. When using a non_blocking socket, nothing needs to be

done, but select() can be used to check for the required condition. If the underlying I/O is blocking, wolfSSL_connect_cert() will only return once the peer's certificate chain has been received.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_get_error
- wolfSSL_connect
- wolfSSL_accept

Return:

- SSL_SUCCESS upon success.
- SSL_FAILURE will be returned if the SSL session parameter is NULL.
- SSL_FATAL_ERROR will be returned if an error occurred. To get a more detailed error code, call wolfSSL_get_error().

Example

```
int ret = 0;
int err = 0;
WOLFSSL* ssl;
char buffer[80];
...
ret = wolfSSL_connect_cert(ssl);
if (ret != SSL_SUCCESS) {
    err = wolfSSL_get_error(ssl, ret);
    printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
}
```

C.58.4.244 function wolfSSL_d2i_PKCS12_bio

```
WC_PKCS12 * wolfSSL_d2i_PKCS12_bio(
    WOLFSSL_BIO * bio,
    WC_PKCS12 ** pkcs12
)
```

wolfSSL_d2i_PKCS12_bio (d2i_PKCS12_bio) copies in the PKCS12 information from WOLFSSL_BIO to the structure WC_PKCS12. The information is divided up in the structure as a list of Content Infos along with a structure to hold optional MAC information. After the information has been divided into chunks (but not decrypted) in the structure WC_PKCS12, it can then be parsed and decrypted by calling.

Parameters:

- **bio** WOLFSSL_BIO structure to read PKCS12 buffer from.
- **pkcs12** WC_PKCS12 structure pointer for new PKCS12 structure created. Can be NULL

See:

- wolfSSL_PKCS12_parse
- wc_PKCS12_free

Return:

- WC_PKCS12 pointer to a WC_PKCS12 structure.
- Failure If function failed it will return NULL.

Example

```
WC_PKCS12* pkcs;
WOLFSSL_BIO* bio;
WOLFSSL_X509* cert;
WOLFSSL_EVP_PKEY* pkey;
STACK_OF(X509) certs;
//bio loads in PKCS12 file
wolfSSL_d2i_PKCS12_bio(bio, &pkcs);
wolfSSL_PKCS12_parse(pkcs, “a password”, &pkey, &cert, &certs)
wc_PKCS12_free(pkcs)
//use cert, pkey, and optionally certs stack
```

C.58.4.245 function wolfSSL_i2d_PKCS12_bio

```
WC_PKCS12 * wolfSSL_i2d_PKCS12_bio(
    WOLFSSL_BIO * bio,
    WC_PKCS12 * pkcs12
)
```

wolfSSL_i2d_PKCS12_bio (i2d_PKCS12_bio) copies in the cert information from the structure WC_PKCS12 to WOLFSSL_BIO.

Parameters:

- **bio** WOLFSSL_BIO structure to write PKCS12 buffer to.
- **pkcs12** WC_PKCS12 structure for PKCS12 structure as input.

See:

- wolfSSL_PKCS12_parse
- wc_PKCS12_free

Return:

- 1 for success.
- Failure 0.

Example

```
WC_PKCS12 pkcs12;
FILE *f;
byte buffer[5300];
char file[] = "./test.p12";
int bytes;
WOLFSSL_BIO* bio;
pkcs12 = wc_PKCS12_new();
f = fopen(file, "rb");
bytes = (int)fread(buffer, 1, sizeof(buffer), f);
fclose(f);
//convert the DER file into an internal structure
wc_d2i_PKCS12(buffer, bytes, pkcs12);
bio = wolfSSL_BIO_new(wolfSSL_BIO_s_mem());
//convert PKCS12 structure into bio
wolfSSL_i2d_PKCS12_bio(bio, pkcs12);
wc_PKCS12_free(pkcs)
//use bio
```

C.58.4.246 function wolfSSL_PKCS12_parse

```
int wolfSSL_PKCS12_parse(
    WC_PKCS12 * pkcs12,
    const char * psw,
    WOLFSSL_EVP_PKEY ** pkey,
    WOLFSSL_X509 ** cert,
    WOLF_STACK_OF(WOLFSSL_X509) ** ca
)
```

PKCS12 can be enabled with adding –enable-opensslextra to the configure command. It can use triple DES and RC4 for decryption so would recommend also enabling these features when enabling opensslextra (–enable-des3 –enable-arc4). wolfSSL does not currently support RC2 so decryption with RC2 is currently not available. This may be noticeable with default encryption schemes used by OpenSSL command line to create .p12 files. wolfSSL_PKCS12_parse (PKCS12_parse). The first thing this function does is check the MAC is correct if present. If the MAC fails then the function returns and does not try to decrypt any of the stored Content Infos. This function then parses through each Content Info looking for a bag type, if the bag type is known it is decrypted as needed and either stored in the list of certificates being built or as a key found. After parsing through all bags the key found is then compared with the certificate list until a matching pair is found. This matching pair is then returned as the key and certificate, optionally the certificate list found is returned as a STACK_OF certificates. At the moment a CRL, Secret or SafeContents bag will be skipped over and not parsed. It can be seen if these or other “Unknown” bags are skipped over by viewing the debug print out. Additional attributes such as friendly name are skipped over when parsing a PKCS12 file.

Parameters:

- **pkcs12** WC_PKCS12 structure to parse.
- **paswd** password for decrypting PKCS12.
- **pkey** structure to hold private key decoded from PKCS12.
- **cert** structure to hold certificate decoded from PKCS12.
- **stack** optional stack of extra certificates.

See:

- wolfSSL_d2i_PKCS12_bio
- wc_PKCS12_free

Return:

- SSL_SUCCESS On successfully parsing PKCS12.
- SSL_FAILURE If an error case was encountered.

Example

```
WC_PKCS12* pkcs;
WOLFSSL_BIO* bio;
WOLFSSL_X509* cert;
WOLFSSL_EVP_PKEY* pkey;
STACK_OF(X509) certs;
//bio loads in PKCS12 file
wolfSSL_d2i_PKCS12_bio(bio, &pkcs);
wolfSSL_PKCS12_parse(pkcs, “a password”, &pkey, &cert, &certs)
wc_PKCS12_free(pkcs)
//use cert, pkey, and optionally certs stack
```

C.58.4.247 function wolfSSL_SetTmpDH

```
int wolfSSL_SetTmpDH(
    WOLFSSL * ssl,
    const unsigned char * p,
```

```
    int pSz,
    const unsigned char * g,
    int gSz
)
```

Server Diffie-Hellman Ephemeral parameters setting. This function sets up the group parameters to be used if the server negotiates a cipher suite that uses DHE.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **p** Diffie-Hellman prime number parameter.
- **pSz** size of p.
- **g** Diffie-Hellman "generator" parameter.
- **gSz** size of g.

See: SSL_accept

Return:

- SSL_SUCCESS upon success.
- MEMORY_ERROR will be returned if a memory error was encountered.
- SIDE_ERROR will be returned if this function is called on an SSL client instead of an SSL server.

Example

```
WOLFSSL* ssl;
static unsigned char p[] = {...};
static unsigned char g[] = {...};
...
wolfSSL_SetTmpDH(ssl, p, sizeof(p), g, sizeof(g));
```

C.58.4.248 function wolfSSL_SetTmpDH_buffer

```
int wolfSSL_SetTmpDH_buffer(
    WOLFSSL * ssl,
    const unsigned char * b,
    long sz,
    int format
)
```

The function calls the wolfSSL_SetTMpDH_buffer_wrapper, which is a wrapper for Diffie-Hellman parameters.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **buf** allocated buffer passed in from wolfSSL_SetTMpDH_file_wrapper.
- **sz** a long int that holds the size of the file (fname within wolfSSL_SetTmpDH_file_wrapper).
- **format** an integer type passed through from wolfSSL_SetTmpDH_file_wrapper() that is a representation of the certificate format.

See:

- wolfSSL_SetTmpDH_buffer_wrapper
- wc_DhParamsLoad
- wolfSSL_SetTmpDH
- PemToDer
- wolfSSL_CTX_SetTmpDH
- wolfSSL_CTX_SetTmpDH_file

Return:

- SSL_SUCCESS on successful execution.
- SSL_BAD_FILETYPE if the file type is not PEM and is not ASN.1. It will also be returned if the wc_DhParamsLoad does not return normally.
- SSL_NO_PEM_HEADER returns from PemToDer if there is not a PEM header.
- SSL_BAD_FILE returned if there is a file error in PemToDer.
- SSL_FATAL_ERROR returned from PemToDer if there was a copy error.
- MEMORY_E - if there was a memory allocation error.
- BAD_FUNC_ARG returned if the WOLFSSL struct is NULL or if there was otherwise a NULL argument passed to a subroutine.
- DH_KEY_SIZE_E is returned if their is a key size error in wolfSSL_SetTmpDH() or in wolfSSL_CTX_SetTmpDH().
- SIDE_ERROR returned if it is not the server side in wolfSSL_SetTmpDH.

Example

```
Static int wolfSSL_SetTmpDH_file_wrapper(WOLFSSL_CTX* ctx, WOLFSSL* ssl,
Const char* fname, int format);
long sz = 0;
byte* myBuffer = staticBuffer[FILE_BUFFER_SIZE];
…
if(ssl)
ret = wolfSSL_SetTmpDH_buffer(ssl, myBuffer, sz, format);
```

C.58.4.249 function wolfSSL_SetTmpDH_file

```
int wolfSSL_SetTmpDH_file(
    WOLFSSL * ssl,
    const char * f,
    int format
)
```

This function calls wolfSSL_SetTmpDH_file_wrapper to set server Diffie-Hellman parameters.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **fname** a constant char pointer holding the certificate.
- **format** an integer type that holds the format of the certification.

See:

- wolfSSL_CTX_SetTmpDH_file
- wolfSSL_SetTmpDH_file_wrapper
- wolfSSL_SetTmpDH_buffer
- wolfSSL_CTX_SetTmpDH_buffer
- wolfSSL_SetTmpDH_buffer_wrapper
- wolfSSL_SetTmpDH
- wolfSSL_CTX_SetTmpDH

Return:

- SSL_SUCCESS returned on successful completion of this function and its subroutines.
- MEMORY_E returned if a memory allocation failed in this function or a subroutine.
- SIDE_ERROR if the side member of the Options structure found in the WOLFSSL struct is not the server side.
- SSL_BAD_FILETYPE returns if the certificate fails a set of checks.

- DH_KEY_SIZE_E returned if the DH parameter's key size is less than the value of the minDhKeySz member in the WOLFSSL struct.
- DH_KEY_SIZE_E returned if the DH parameter's key size is greater than the value of the maxDhKeySz member in the WOLFSSL struct.
- BAD_FUNC_ARG returns if an argument value is NULL that is not permitted such as, the WOLFSSL structure.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
const char* dhParam;
…
AssertIntNE(SSL_SUCCESS,
wolfSSL_SetTmpDH_file(ssl, dhParam, SSL_FILETYPE_PEM));
```

C.58.4.250 function wolfSSL_CTX_SetTmpDH

```
int wolfSSL_CTX_SetTmpDH(
    WOLFSSL_CTX * ctx,
    const unsigned char * p,
    int pSz,
    const unsigned char * g,
    int gSz
)
```

Sets the parameters for the server CTX Diffie-Hellman.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **p** a constant unsigned char pointer loaded into the buffer member of the serverDH_P struct.
- **pSz** an int type representing the size of p, initialized to MAX_DH_SIZE.
- **g** a constant unsigned char pointer loaded into the buffer member of the serverDH_G struct.
- **gSz** an int type representing the size of g, initialized to MAX_DH_SIZE.

See:

- wolfSSL_SetTmpDH
- wc_DhParamsLoad

Return:

- SSL_SUCCESS returned if the function and all subroutines return without error.
- BAD_FUNC_ARG returned if the CTX, p or g parameters are NULL.
- DH_KEY_SIZE_E returned if the DH parameter's key size is less than the value of the minDhKeySz member of the WOLFSSL_CTX struct.
- DH_KEY_SIZE_E returned if the DH parameter's key size is greater than the value of the maxDhKeySz member of the WOLFSSL_CTX struct.
- MEMORY_E returned if the allocation of memory failed in this function or a subroutine.

Exmaple

```
WOLFSSL_CTX* ctx =  WOLFSSL_CTX_new( protocol );
byte* p;
byte* g;
word32 pSz = (word32)sizeof(p)/sizeof(byte);
word32 gSz = (word32)sizeof(g)/sizeof(byte);
…
int ret =  wolfSSL_CTX_SetTmpDH(ctx, p, pSz, g, gSz);
```

```
if(ret != SSL_SUCCESS){
    // Failure case
}
```

C.58.4.251 function wolfSSL_CTX_SetTmpDH_buffer

```
int wolfSSL_CTX_SetTmpDH_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * b,
    long sz,
    int format
)
```

A wrapper function that calls wolfSSL_SetTmpDH_buffer_wrapper.

Parameters:

- **ctx** a pointer to a WOLFSSL structure, created using wolfSSL_CTX_new().
- **buf** a pointer to a constant unsigned char type that is allocated as the buffer and passed through to wolfSSL_SetTmpDH_buffer_wrapper.
- **sz** a long integer type that is derived from the fname parameter in wolfSSL_SetTmpDH_file_wrapper().
- **format** an integer type passed through from wolfSSL_SetTmpDH_file_wrapper().

See:

- wolfSSL_SetTmpDH_buffer_wrapper
- wolfSSL_SetTMpDH_buffer
- wolfSSL_SetTmpDH_file_wrapper
- wolfSSL_CTX_SetTmpDH_file

Return:

- 0 returned for a successful execution.
- BAD_FUNC_ARG returned if the ctx or buf parameters are NULL.
- MEMORY_E if there is a memory allocation error.
- SSL_BAD_FILETYPE returned if format is not correct.

Example

```
static int wolfSSL_SetTmpDH_file_wrapper(WOLFSSL_CTX* ctx, WOLFSSL* ssl,
Const char* fname, int format);
#ifdef WOLFSSL_SMALL_STACK
byte staticBuffer[1]; // force heap usage
#else
byte* staticBuffer;
long sz = 0;
…
if(ssl){
    ret = wolfSSL_SetTmpDH_buffer(ssl, myBuffer, sz, format);
} else {
ret = wolfSSL_CTX_SetTmpDH_buffer(ctx, myBuffer, sz, format);
}
```

C.58.4.252 function wolfSSL_CTX_SetTmpDH_file

```
int wolfSSL_CTX_SetTmpDH_file(
    WOLFSSL_CTX * ctx,
    const char * f,
```

```
    int format
)
```

The function calls wolfSSL_SetTmpDH_file_wrapper to set the server Diffie-Hellman parameters.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **fname** a constant character pointer to a certificate file.
- **format** an integer type passed through from wolfSSL_SetTmpDH_file_wrapper() that is a representation of the certificate format.

See:

- wolfSSL_SetTmpDH_buffer_wrapper
- wolfSSL_SetTmpDH
- wolfSSL_CTX_SetTmpDH
- wolfSSL_SetTmpDH_buffer
- wolfSSL_CTX_SetTmpDH_buffer
- wolfSSL_SetTmpDH_file_wrapper
- AllocDer
- PemToDer

Return:

- SSL_SUCCESS returned if the wolfSSL_SetTmpDH_file_wrapper or any of its subroutines return successfully.
- MEMORY_E returned if an allocation of dynamic memory fails in a subroutine.
- BAD_FUNC_ARG returned if the ctx or fname parameters are NULL or if a subroutine is passed a NULL argument.
- SSL_BAD_FILE returned if the certificate file is unable to open or if the a set of checks on the file fail from wolfSSL_SetTmpDH_file_wrapper.
- SSL_BAD_FILETYPE returned if the format is not PEM or ASN.1 from wolfSSL_SetTmpDH_buffer_wrapper().
- DH_KEY_SIZE_E returned if the DH parameter's key size is less than the value of the minDhKeySz member of the WOLFSSL_CTX struct.
- DH_KEY_SIZE_E returned if the DH parameter's key size is greater than the value of the maxDhKeySz member of the WOLFSSL_CTX struct.
- SIDE_ERROR returned in wolfSSL_SetTmpDH() if the side is not the server end.
- SSL_NO_PEM_HEADER returned from PemToDer if there is no PEM header.
- SSL_FATAL_ERROR returned from PemToDer if there is a memory copy failure.

Example

```
#define dhParam     "certs/dh2048.pem"
#DEFINE aSSERTiNTne(x, y)     AssertInt(x, y, !=, ==)
WOLFSSL_CTX* ctx;
…
AssertNotNull(ctx = wolfSSL_CTX_new(wolfSSLv23_client_method()))
…
AssertIntNE(SSL_SUCCESS, wolfSSL_CTX_SetTmpDH_file(NULL, dhParam,
SSL_FILETYPE_PEM));
```

C.58.4.253 function wolfSSL_CTX_SetMinDhKey_Sz

```
int wolfSSL_CTX_SetMinDhKey_Sz(
    WOLFSSL_CTX * ctx,
    word16 keySz_bits
)
```

This function sets the minimum size (in bits) of the Diffie Hellman key size by accessing the minDhKeySz member in the WOLFSSL_CTX structure.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **keySz_bits** a word16 type used to set the minimum DH key size in bits. The WOLFSSL_CTX struct holds this information in the minDhKeySz member.

See:

- wolfSSL_SetMinDhKey_Sz
- wolfSSL_CTX_SetMaxDhKey_Sz
- wolfSSL_SetMaxDhKey_Sz
- wolfSSL_GetDhKey_Sz
- wolfSSL_CTX_SetTMpDH_file

Return:

- SSL_SUCCESS returned if the function completes successfully.
- BAD_FUNC_ARG returned if the WOLFSSL_CTX struct is NULL or if the keySz_bits is greater than 16,000 or not divisible by 8.

Example

```
public static int CTX_SetMinDhKey_Sz(IntPtr ctx, short minDhKey){
…
return wolfSSL_CTX_SetMinDhKey_Sz(local_ctx, minDhKeyBits);
```

C.58.4.254 function wolfSSL_SetMinDhKey_Sz

```
int wolfSSL_SetMinDhKey_Sz(
    WOLFSSL * ssl,
    word16 keySz_bits
)
```

Sets the minimum size (in bits) for a Diffie-Hellman key in the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **keySz_bits** a word16 type used to set the minimum DH key size in bits. The WOLFSSL_CTX struct holds this information in the minDhKeySz member.

See:

- wolfSSL_CTX_SetMinDhKey_Sz
- wolfSSL_GetDhKey_Sz

Return:

- SSL_SUCCESS the minimum size was successfully set.
- BAD_FUNC_ARG the WOLFSSL structure was NULL or if the keySz_bits is greater than 16,000 or not divisible by 8.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
word16 keySz_bits;
...
if(wolfSSL_SetMinDhKey_Sz(ssl, keySz_bits) != SSL_SUCCESS){
    // Failed to set.
}
```

C.58.4.255 function wolfSSL_CTX_SetMaxDhKey_Sz

```
int wolfSSL_CTX_SetMaxDhKey_Sz(
    WOLFSSL_CTX * ctx,
    word16 keySz_bits
)
```

This function sets the maximum size (in bits) of the Diffie Hellman key size by accessing the maxDhKeySz member in the WOLFSSL_CTX structure.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **keySz_bits** a word16 type used to set the maximum DH key size in bits. The WOLFSSL_CTX struct holds this information in the maxDhKeySz member.

See:

- wolfSSL_SetMinDhKey_Sz
- wolfSSL_CTX_SetMinDhKey_Sz
- wolfSSL_SetMaxDhKey_Sz
- wolfSSL_GetDhKey_Sz
- wolfSSL_CTX_SetTMpDH_file

Return:

- SSL_SUCCESS returned if the function completes successfully.
- BAD_FUNC_ARG returned if the WOLFSSL_CTX struct is NULL or if the keySz_bits is greater than 16,000 or not divisible by 8.

Example

```
public static int CTX_SetMaxDhKey_Sz(IntPtr ctx, short maxDhKey){
…
return wolfSSL_CTX_SetMaxDhKey_Sz(local_ctx, keySz_bits);
```

C.58.4.256 function wolfSSL_SetMaxDhKey_Sz

```
int wolfSSL_SetMaxDhKey_Sz(
    WOLFSSL * ssl,
    word16 keySz_bits
)
```

Sets the maximum size (in bits) for a Diffie-Hellman key in the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **keySz** a word16 type representing the bit size of the maximum DH key.

See:

- wolfSSL_CTX_SetMaxDhKey_Sz
- wolfSSL_GetDhKey_Sz

Return:

- SSL_SUCCESS the maximum size was successfully set.
- BAD_FUNC_ARG the WOLFSSL structure was NULL or the keySz parameter was greater than the allowable size or not divisible by 8.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
word16 keySz;
...
if(wolfSSL_SetMaxDhKey(ssl, keySz) != SSL_SUCCESS){
    // Failed to set.
}
```

C.58.4.257 function wolfSSL_GetDhKey_Sz

```
int wolfSSL_GetDhKey_Sz(
    WOLFSSL * ssl
)
```

Returns the value of dhKeySz (in bits) that is a member of the options structure. This value represents the Diffie-Hellman key size in bytes.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_SetMinDhKey_sz
- wolfSSL_CTX_SetMinDhKey_Sz
- wolfSSL_CTX_SetTmpDH
- wolfSSL_SetTmpDH
- wolfSSL_CTX_SetTmpDH_file

Return:

- dhKeySz returns the value held in ssl->options.dhKeySz which is an integer value representing a size in bits.
- BAD_FUNC_ARG returns if the WOLFSSL struct is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
int dhKeySz;
...
dhKeySz = wolfSSL_GetDhKey_Sz(ssl);

if(dhKeySz == BAD_FUNC_ARG || dhKeySz <= 0){
    // Failure case
} else {
    // dhKeySz holds the size of the key.
}
```

C.58.4.258 function wolfSSL_CTX_SetMinRsaKey_Sz

```
int wolfSSL_CTX_SetMinRsaKey_Sz(
    WOLFSSL_CTX * ctx,
    short keySz
)
```

Sets the minimum RSA key size in both the WOLFSSL_CTX structure and the WOLFSSL_CERT_MANAGER structure.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **keySz** a short integer type stored in minRsaKeySz in the ctx structure and the cm structure converted to bytes.

See: wolfSSL_SetMinRsaKey_Sz

Return:

- SSL_SUCCESS returned on successful execution of the function.
- BAD_FUNC_ARG returned if the ctx structure is NULL or the keySz is less than zero or not divisible by 8.

Example

```
WOLFSSL_CTX* ctx = SSL_CTX_new(method);
(void)minDhKeyBits;
ourCert = myoptarg;
…
minDhKeyBits = atoi(myoptarg);
…
if(wolfSSL_CTX_SetMinRsaKey_Sz(ctx, minRsaKeyBits) != SSL_SUCCESS){
…
```

C.58.4.259 function wolfSSL_SetMinRsaKey_Sz

```
int wolfSSL_SetMinRsaKey_Sz(
    WOLFSSL * ssl,
    short keySz
)
```

Sets the minimum allowable key size in bits for RSA located in the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **keySz** a short integer value representing the the minimum key in bits.

See: wolfSSL_CTX_SetMinRsaKey_Sz

Return:

- SSL_SUCCESS the minimum was set successfully.
- BAD_FUNC_ARG returned if the ssl structure is NULL or if the ksySz is less than zero or not divisible by 8.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
short keySz;
…

int isSet =  wolfSSL_SetMinRsaKey_Sz(ssl, keySz);
if(isSet != SSL_SUCCESS){
    Failed to set.
}
```

C.58.4.260 function wolfSSL_CTX_SetMinEccKey_Sz

```
int wolfSSL_CTX_SetMinEccKey_Sz(
    WOLFSSL_CTX * ctx,
```

```
    short keySz
)
```

Sets the minimum size in bits for the ECC key in the WOLF_CTX structure and the WOLFSSL_CERT_MANAGER structure.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **keySz** a short integer type that represents the minimum ECC key size in bits.

See: wolfSSL_SetMinEccKey_Sz

Return:

- SSL_SUCCESS returned for a successful execution and the minEccKeySz member is set.
- BAD_FUNC_ARG returned if the WOLFSSL_CTX struct is NULL or if the keySz is negative or not divisible by 8.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
short keySz; // minimum key size
…
if(wolfSSL_CTX_SetMinEccKey(ctx, keySz) != SSL_SUCCESS){
    // Failed to set min key size
}
```

C.58.4.261 function wolfSSL_SetMinEccKey_Sz

```
int wolfSSL_SetMinEccKey_Sz(
    WOLFSSL * ssl,
    short keySz
)
```

Sets the value of the minEccKeySz member of the options structure. The options struct is a member of the WOLFSSL structure and is accessed through the ssl parameter.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **keySz** value used to set the minimum ECC key size. Sets value in the options structure.

See:

- wolfSSL_CTX_SetMinEccKey_Sz
- wolfSSL_CTX_SetMinRsaKey_Sz
- wolfSSL_SetMinRsaKey_Sz

Return:

- SSL_SUCCESS if the function successfully set the minEccKeySz member of the options structure.
- BAD_FUNC_ARG if the WOLFSSL_CTX structure is NULL or if the key size (keySz) is less than 0 (zero) or not divisible by 8.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx); // New session
short keySz = 999; // should be set to min key size allowable
...
if(wolfSSL_SetMinEccKey_Sz(ssl, keySz) != SSL_SUCCESS){
    // Failure case.
}
```

C.58.4.262 function wolfSSL_make_eap_keys

```
int wolfSSL_make_eap_keys(
    WOLFSSL * ssl,
    void * key,
    unsigned int len,
    const char * label
)
```

This function is used by EAP_TLS and EAP-TTLS to derive keying material from the master secret.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **key** a void pointer variable that will hold the result of the p_hash function.
- **len** an unsigned integer that represents the length of the key variable.
- **label** a constant char pointer that is copied from in wc_PRF().

See:

- wc_PRF
- wc_HmacFinal
- wc_HmacUpdate

Return:

- BUFFER_E returned if the actual size of the buffer exceeds the maximum size allowable.
- MEMORY_E returned if there is an error with memory allocation.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);;
void* key;
unsigned int len;
const char* label;
…
return wolfSSL_make_eap_keys(ssl, key, len, label);
```

C.58.4.263 function wolfSSL_writev

```
int wolfSSL_writev(
    WOLFSSL * ssl,
    const struct iovec * iov,
    int iovcnt
)
```

Simulates writev semantics but doesn't actually do block at a time because of SSL_write() behavior and because front adds may be small. Makes porting into software that uses writev easier.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **iov** array of I/O vectors to write
- **iovcnt** number of vectors in iov array.

See: wolfSSL_write

Return:

- 0 the number of bytes written upon success.
- 0 will be returned upon failure. Call wolfSSL_get_error() for the specific error code.

- MEMORY_ERROR will be returned if a memory error was encountered.
- BAD_FUNC_ARG will be returned when ssl is NULL, iovcnt is negative, or iov is NULL with a non-zero iovcnt.
- SSL_FATAL_ERROR will be returned upon failure when either an error occurred or, when using non-blocking sockets, the SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE error was received and and the application needs to call wolfSSL_write() again. Use wolfSSL_get_error() to get a specific error code.
- BUFFER_E will be returned when the total length of the segments overflows.

Example

```
WOLFSSL* ssl = 0;
char *bufA = “hello\n”;
char *bufB = “hello world\n”;
int iovcnt;
struct iovec iov[2];

iov[0].iov_base = buffA;
iov[0].iov_len = strlen(buffA);
iov[1].iov_base = buffB;
iov[1].iov_len = strlen(buffB);
iovcnt = 2;
...
ret = wolfSSL_writev(ssl, iov, iovcnt);
// wrote “ret” bytes, or error if <= 0.
```

C.58.4.264 function wolfSSL_CTX_UnloadCAs

```
int wolfSSL_CTX_UnloadCAs(
    WOLFSSL_CTX * ctx
)
```

This function unloads the CA signer list and frees the whole signer table.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

See:

- wolfSSL_CertManagerUnloadCAs
- LockMutex
- UnlockMutex

Return:

- SSL_SUCCESS returned on successful execution of the function.
- BAD_FUNC_ARG returned if the WOLFSSL_CTX struct is NULL or there are otherwise unpermitted argument values passed in a subroutine.
- BAD_MUTEX_E returned if there was a mutex error. The LockMutex() did not return 0.

Example

```
WOLFSSL_METHOD method = wolfTLSv1_2_client_method();
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(method);
…
if(wolfSSL_CTX_UnloadCAs(ctx) != SSL_SUCCESS){
```

```
    // The function did not unload CAs
}
```

C.58.4.265 function wolfSSL_CTX_UnloadIntermediateCerts

```
int wolfSSL_CTX_UnloadIntermediateCerts(
    WOLFSSL_CTX * ctx
)
```

This function unloads intermediate certificates added to the CA signer list and frees them.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

See:

- wolfSSL_CTX_UnloadCAs
- wolfSSL_CertManagerUnloadIntermediateCerts

Return:

- SSL_SUCCESS returned on successful execution of the function.
- BAD_FUNC_ARG returned if the WOLFSSL_CTX struct is NULL or there are otherwise unpermitted argument values passed in a subroutine.
- BAD_STATE_E returned if the WOLFSSL_CTX has a reference count > 1.
- BAD_MUTEX_E returned if there was a mutex error. The LockMutex() did not return 0.

Example

```
WOLFSSL_METHOD method = wolfTLSv1_2_client_method();
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(method);
…
if(wolfSSL_CTX_UnloadIntermediateCerts(ctx) != NULL){
    // The function did not unload CAs
}
```

C.58.4.266 function wolfSSL_CTX_Unload_trust_peers

```
int wolfSSL_CTX_Unload_trust_peers(
    WOLFSSL_CTX * ctx
)
```

This function is used to unload all previously loaded trusted peer certificates. Feature is enabled by defining the macro WOLFSSL_TRUST_PEER_CERT.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().

See:

- wolfSSL_CTX_trust_peer_buffer
- wolfSSL_CTX_trust_peer_cert

Return:

- SSL_SUCCESS upon success.
- BAD_FUNC_ARG will be returned if ctx is NULL.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- BAD_MUTEX_E will be returned when locking fails.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_Unload_trust_peers(ctx);
if (ret != SSL_SUCCESS) {
    // error unloading trusted peer certs
}
...
```

C.58.4.267 function wolfSSL_CTX_trust_peer_buffer

```
int wolfSSL_CTX_trust_peer_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format
)
```

This function loads a certificate to use for verifying a peer when performing a TLS/SSL handshake. The peer certificate sent during the handshake is compared by using the SKID when available and the signature. If these two things do not match then any loaded CAs are used. Is the same functionality as wolfSSL_CTX_trust_peer_cert except is from a buffer instead of a file. Feature is enabled by defining the macro WOLFSSL_TRUST_PEER_CERT Please see the examples for proper usage.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **buffer** pointer to the buffer containing certificates.
- **sz** length of the buffer input.
- **type** type of certificate being loaded i.e. SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_use_certificate_file
- wolfSSL_CTX_use_PrivateKey_file
- wolfSSL_CTX_use_certificate_chain_file
- wolfSSL_CTX_trust_peer_cert
- wolfSSL_CTX_Unload_trust_peers
- wolfSSL_use_certificate_file
- wolfSSL_use_PrivateKey_file
- wolfSSL_use_certificate_chain_file

Return:

- SSL_SUCCESS upon success
- SSL_FAILURE will be returned if ctx is NULL, or if both file and type are invalid.
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
```

```
ret = wolfSSL_CTX_trust_peer_buffer(ctx, bufferPtr, bufferSz,
SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
// error loading trusted peer cert
}
...
```

C.58.4.268 function wolfSSL_CTX_load_verify_buffer

```
int wolfSSL_CTX_load_verify_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format
)
```

This function loads a CA certificate buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. More than one CA certificate may be loaded per buffer as long as the format is in PEM. Please see the examples for proper usage.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **in** pointer to the CA certificate buffer.
- **sz** size of the input CA certificate buffer, in.
- **format** format of the buffer certificate, either SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CTX_load_verify_locations
- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_certificate_chain_buffer
- wolfSSL_use_certificate_buffer
- wolfSSL_use_PrivateKey_buffer
- wolfSSL_use_certificate_chain_buffer

Return:

- SSL_SUCCESS upon success
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- BUFFER_E will be returned if a chain buffer is bigger than the receiving buffer.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
byte certBuff[...];
long sz = sizeof(certBuff);
...

ret = wolfSSL_CTX_load_verify_buffer(ctx, certBuff, sz, SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
```

```
    // error loading CA certs from buffer
}
...
```

C.58.4.269 function wolfSSL_CTX_load_verify_buffer_ex

```
int wolfSSL_CTX_load_verify_buffer_ex(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format,
    int userChain,
    word32 flags
)
```

This function loads a CA certificate buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. More than one CA certificate may be loaded per buffer as long as the format is in PEM. The _ex version was added in PR 2413 and supports additional arguments for userChain and flags.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **in** pointer to the CA certificate buffer.
- **sz** size of the input CA certificate buffer, in.
- **format** format of the buffer certificate, either SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.
- **userChain** If using format WOLFSSL_FILETYPE_ASN1 this set to non-zero indicates a chain of DER's is being presented.
- **flags** See ssl.h around WOLFSSL_LOAD_VERIFY_DEFAULT_FLAGS.

See:

- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_load_verify_locations
- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_certificate_chain_buffer
- wolfSSL_use_certificate_buffer
- wolfSSL_use_PrivateKey_buffer
- wolfSSL_use_certificate_chain_buffer

Return:

- SSL_SUCCESS upon success
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- BUFFER_E will be returned if a chain buffer is bigger than the receiving buffer.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
byte certBuff[...];
long sz = sizeof(certBuff);
...
```

```
// Example for force loading an expired certificate
ret = wolfSSL_CTX_load_verify_buffer_ex(ctx, certBuff, sz, SSL_FILETYPE_PEM,
    0, (WOLFSSL_LOAD_FLAG_DATE_ERR_OKAY));
if (ret != SSL_SUCCESS) {
    // error loading CA certs from buffer
}
...
```

C.58.4.270 function wolfSSL_CTX_load_verify_chain_buffer_format

```
int wolfSSL_CTX_load_verify_chain_buffer_format(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format
)
```

This function loads a CA certificate chain buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. More than one CA certificate may be loaded per buffer as long as the format is in PEM. Please see the examples for proper usage.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **in** pointer to the CA certificate buffer.
- **sz** size of the input CA certificate buffer, in.
- **format** format of the buffer certificate, either SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CTX_load_verify_locations
- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_certificate_chain_buffer
- wolfSSL_use_certificate_buffer
- wolfSSL_use_PrivateKey_buffer
- wolfSSL_use_certificate_chain_buffer

Return:

- SSL_SUCCESS upon success
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- BUFFER_E will be returned if a chain buffer is bigger than the receiving buffer.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
byte certBuff[...];
long sz = sizeof(certBuff);
...

ret = wolfSSL_CTX_load_verify_chain_buffer_format(ctx,
```

```
                    certBuff, sz, WOLFSSL_FILETYPE_ASN1);
if (ret != SSL_SUCCESS) {
    // error loading CA certs from buffer
}
...
```

C.58.4.271 function wolfSSL_CTX_use_certificate_buffer

```
int wolfSSL_CTX_use_certificate_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format
)
```

This function loads a certificate buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. Please see the examples for proper usage.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **in** the input buffer containing the certificate to be loaded.
- **sz** the size of the input buffer.
- **format** the format of the certificate located in the input buffer (in). Possible values are SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_certificate_chain_buffer
- wolfSSL_use_certificate_buffer
- wolfSSL_use_PrivateKey_buffer
- wolfSSL_use_certificate_chain_buffer

Return:

- SSL_SUCCESS upon success
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
byte certBuff[...];
long sz = sizeof(certBuff);
...
ret = wolfSSL_CTX_use_certificate_buffer(ctx, certBuff, sz, SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading certificate from buffer
}
...
```

C.58.4.272 function wolfSSL_CTX_use_PrivateKey_buffer

```
int wolfSSL_CTX_use_PrivateKey_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format
)
```

This function loads a private key buffer into the SSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1or SSL_FILETYPE_PEM. Please see the examples for proper usage.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **in** the input buffer containing the private key to be loaded.
- **sz** the size of the input buffer.
- **format** the format of the private key located in the input buffer (in). Possible values are SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_CTX_use_certificate_chain_buffer
- wolfSSL_use_certificate_buffer
- wolfSSL_use_PrivateKey_buffer
- wolfSSL_use_certificate_chain_buffer

Return:

- SSL_SUCCESS upon success
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- NO_PASSWORD will be returned if the key file is encrypted but no password is provided.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
byte keyBuff[...];
long sz = sizeof(certBuff);
...
ret = wolfSSL_CTX_use_PrivateKey_buffer(ctx, keyBuff, sz, SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading private key from buffer
}
...
```

C.58.4.273 function wolfSSL_CTX_use_certificate_chain_buffer

```
int wolfSSL_CTX_use_certificate_chain_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
```

```
    long sz
)
```

This function loads a certificate chain buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. The buffer must be in PEM format and start with the subject's certificate, ending with the root certificate. Please see the examples for proper usage.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **in** the input buffer containing the PEM-formatted certificate chain to be loaded.
- **sz** the size of the input buffer.

See:

- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_use_certificate_buffer
- wolfSSL_use_PrivateKey_buffer
- wolfSSL_use_certificate_chain_buffer

Return:

- SSL_SUCCESS upon success
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- BUFFER_E will be returned if a chain buffer is bigger than the receiving buffer.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
byte certChainBuff[...];
long sz = sizeof(certBuff);
...
ret = wolfSSL_CTX_use_certificate_chain_buffer(ctx, certChainBuff, sz);
if (ret != SSL_SUCCESS) {
    // error loading certificate chain from buffer
}
...
```

C.58.4.274 function wolfSSL_use_certificate_buffer

```
int wolfSSL_use_certificate_buffer(
    WOLFSSL * ssl,
    const unsigned char * in,
    long sz,
    int format
)
```

This function loads a certificate buffer into the WOLFSSL object. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. Please see the examples for proper usage.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **in** buffer containing certificate to load.
- **sz** size of the certificate located in buffer.
- **format** format of the certificate to be loaded. Possible values are SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_certificate_chain_buffer
- wolfSSL_use_PrivateKey_buffer
- wolfSSL_use_certificate_chain_buffer

Return:

- SSL_SUCCESS upon success.
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.

Example

```
int ret;
byte certBuff[...];
WOLFSSL* ssl = 0;
long buffSz = sizeof(certBuff);
...

ret = wolfSSL_use_certificate_buffer(ssl, certBuff, buffSz, SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // failed to load certificate from buffer
}
```

C.58.4.275 function wolfSSL_use_PrivateKey_buffer

```
int wolfSSL_use_PrivateKey_buffer(
    WOLFSSL * ssl,
    const unsigned char * in,
    long sz,
    int format
)
```

This function loads a private key buffer into the WOLFSSL object. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. Please see the examples for proper usage.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **in** buffer containing private key to load.
- **sz** size of the private key located in buffer.
- **format** format of the private key to be loaded. Possible values are SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_use_PrivateKey
- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_certificate_chain_buffer
- wolfSSL_use_certificate_buffer
- wolfSSL_use_certificate_chain_buffer

Return:

- SSL_SUCCESS upon success.
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- NO_PASSWORD will be returned if the key file is encrypted but no password is provided.

Example

```
int ret;
byte keyBuff[...];
WOLFSSL* ssl = 0;
long buffSz = sizeof(certBuff);
...
ret = wolfSSL_use_PrivateKey_buffer(ssl, keyBuff, buffSz, SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // failed to load private key from buffer
}
```

C.58.4.276 function wolfSSL_use_certificate_chain_buffer

```
int wolfSSL_use_certificate_chain_buffer(
    WOLFSSL * ssl,
    const unsigned char * in,
    long sz
)
```

This function loads a certificate chain buffer into the WOLFSSL object. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. The buffer must be in PEM format and start with the subject's certificate, ending with the root certificate. Please see the examples for proper usage.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **in** buffer containing certificate to load.
- **sz** size of the certificate located in buffer.

See:

- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_certificate_chain_buffer
- wolfSSL_use_certificate_buffer
- wolfSSL_use_PrivateKey_buffer

Return:

- SSL_SUCCES upon success.

- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- BUFFER_E will be returned if a chain buffer is bigger than the receiving buffer.

Example

```
int ret;
byte certChainBuff[...];
WOLFSSL* ssl = 0;
long buffSz = sizeof(certBuff);
...
ret = wolfSSL_use_certificate_chain_buffer(ssl, certChainBuff, buffSz);
if (ret != SSL_SUCCESS) {
    // failed to load certificate chain from buffer
}
```

C.58.4.277 function wolfSSL_UnloadCertsKeys

```
int wolfSSL_UnloadCertsKeys(
    WOLFSSL * ssl
)
```

This function unloads any certificates or keys that SSL owns.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_CTX_UnloadCAs

Return:

- SSL_SUCCESS - returned if the function executed successfully.
- BAD_FUNC_ARG - returned if the WOLFSSL object is NULL.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
...
int unloadKeys = wolfSSL_UnloadCertsKeys(ssl);
if(unloadKeys != SSL_SUCCESS){
    // Failure case.
}
```

C.58.4.278 function wolfSSL_CTX_set_group_messages

```
int wolfSSL_CTX_set_group_messages(
    WOLFSSL_CTX * ctx
)
```

This function turns on grouping of handshake messages where possible.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().

See:

- wolfSSL_set_group_messages
- wolfSSL_CTX_clear_group_messages

- wolfSSL_clear_group_messages
- wolfSSL_CTX_new

Return:

- SSL_SUCCESS will be returned upon success.
- BAD_FUNC_ARG will be returned if the input context is null.

Example

```
WOLFSSL_CTX* ctx = 0;
...
ret = wolfSSL_CTX_set_group_messages(ctx);
if (ret != SSL_SUCCESS) {
    // failed to set handshake message grouping
}
```

C.58.4.279 function wolfSSL_set_group_messages

```
int wolfSSL_set_group_messages(
    WOLFSSL * ssl
)
```

This function turns on grouping of handshake messages where possible.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See:

- wolfSSL_CTX_set_group_messages
- wolfSSL_clear_group_messages
- wolfSSL_CTX_clear_group_messages
- wolfSSL_new

Return:

- SSL_SUCCESS will be returned upon success.
- BAD_FUNC_ARG will be returned if the input context is null.

Example

```
WOLFSSL* ssl = 0;
...
ret = wolfSSL_set_group_messages(ssl);
if (ret != SSL_SUCCESS) {
// failed to set handshake message grouping
}
```

C.58.4.280 function wolfSSL_SetFuzzerCb

```
void wolfSSL_SetFuzzerCb(
    WOLFSSL * ssl,
    CallbackFuzzer cbf,
    void * fCtx
)
```

This function sets the fuzzer callback.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **cbf** a CallbackFuzzer type that is a function pointer of the form: int (*CallbackFuzzer)(WOLFSSL* ssl, const unsigned char* buf, int sz, int type, void* fuzzCtx);
- **fCtx** a void pointer type that will be set to the fuzzerCtx member of the WOLFSSL structure.

See: CallbackFuzzer

Return: none No returns.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
void* fCtx;

int callbackFuzzerCB(WOLFSSL* ssl, const unsigned char* buf, int sz,
            int type, void* fuzzCtx){
// function definition
}
…
wolfSSL_SetFuzzerCb(ssl, callbackFuzzerCB, fCtx);
```

C.58.4.281 function wolfSSL_DTLS_SetCookieSecret

```
int wolfSSL_DTLS_SetCookieSecret(
    WOLFSSL * ssl,
    const byte * secret,
    word32 secretSz
)
```

This function sets a new dtls cookie secret.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **secret** a constant byte pointer representing the secret buffer. Passing NULL indicates to generate a new random secret.
- **secretSz** the size of the buffer. Passing 0 with a NULL secret generates a secret of the default size (COOKIE_SECRET_SZ); passing 0 with a non-NULL secret returns BAD_FUNC_ARG.

See:

- ForceZero
- wc_RNG_GenerateBlock

Return:

- 0 returned if the function executed without an error.
- BAD_FUNC_ARG returned if there was an argument passed to the function with an unacceptable value.
- MEMORY_ERROR returned if there was a problem allocating memory for a new cookie secret.
- BAD_STATE_E returned if secret is NULL and the session's RNG has been released, so no secret can be generated.
- Another -ve value when a new secret could not be generated. The rotation does not happen and the secret already in use is kept, so the server carries on issuing and verifying cookies under it.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
const* byte secret;
```

```
word32 secretSz; // size of secret
…
if(!wolfSSL_DTLS_SetCookieSecret(ssl, secret, secretSz)){
    // Code block for failure to set DTLS cookie secret
} else {
    // Success! Cookie secret is set.
}
```

C.58.4.282 function wolfSSL_DTLS_SetCookieSecretSecondary

```
int wolfSSL_DTLS_SetCookieSecretSecondary(
    WOLFSSL * ssl,
    const byte * secret,
    word32 secretSz
)
```

This function sets a secondary DTLS 1.2 cookie secret used only when verifying a received HelloVerifyRequest cookie, and only if the primary secret (set with wolfSSL_DTLS_SetCookieSecret()) fails to verify it. This lets an application rotate the cookie secret on a stateless server without rejecting clients whose cookie was issued under the previous secret: install the new secret as the primary and the previous secret here for an overlap window. The secondary secret is never used to issue cookies. It is the DTLS 1.2 counterpart of wolfSSL_set_hrr_cookie_secret_secondary().

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **secret** a constant byte pointer representing the secret buffer. Passing NULL (or a secretSz of 0) clears any previously set secondary secret.
- **secretSz** the size of the buffer.

See: wolfSSL_DTLS_SetCookieSecret

Return:

- 0 returned if the function executed without an error.
- BAD_FUNC_ARG returned if ssl is NULL.
- MEMORY_ERROR returned if there was a problem allocating memory for the secondary cookie secret. The secondary secret already in use, if any, is kept.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
const byte* oldSecret;
word32 oldSecretSz; // size of oldSecret
...
if(wolfSSL_DTLS_SetCookieSecretSecondary(ssl, oldSecret, oldSecretSz) != 0){
    // Code block for failure to set secondary DTLS cookie secret
} else {
    // Success! Secondary cookie secret is set.
}
```

C.58.4.283 function wolfSSL_GetRNG

```
WC_RNG * wolfSSL_GetRNG(
    WOLFSSL * ssl
)
```

This function retrieves the random number.

Parameters:

- **ssl** pointer to a SSL object, created with wolfSSL_new().

See: wolfSSL_CTX_new_rng

Return:

- rng upon success.
- NULL if ssl is NULL.

Example

```
WOLFSSL* ssl;

wolfSSL_GetRNG(ssl);
```

C.58.4.284 function wolfSSL_CTX_SetMinVersion

```
int wolfSSL_CTX_SetMinVersion(
    WOLFSSL_CTX * ctx,
    int version
)
```

This function sets the minimum downgrade version allowed. Applicable only when the connection allows downgrade using (wolfSSLv23_client_method or wolfSSLv23_server_method).

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **version** an integer representation of the version to be set as the minimum: WOLFSSL_SSLV3 = 0, WOLFSSL_TLSV1 = 1, WOLFSSL_TLSV1_1 = 2 or WOLFSSL_TLSV1_2 = 3.

See: SetMinVersionHelper

Return:

- SSL_SUCCESS returned if the function returned without error and the minimum version is set.
- BAD_FUNC_ARG returned if the WOLFSSL_CTX structure was NULL or if the minimum version is not supported.

Example

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new( protocol method );
WOLFSSL* ssl = WOLFSSL_new(ctx);
int version; // macrop representation
…
if(wolfSSL_CTX_SetMinVersion(ssl->ctx, version) != SSL_SUCCESS){
    // Failed to set min version
}
```

C.58.4.285 function wolfSSL_SetMinVersion

```
int wolfSSL_SetMinVersion(
    WOLFSSL * ssl,
    int version
)
```

This function sets the minimum downgrade version allowed. Applicable only when the connection allows downgrade using (wolfSSLv23_client_method or wolfSSLv23_server_method).

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **version** an integer representation of the version to be set as the minimum: WOLFSSL_SSLV3 = 0, WOLFSSL_TLSV1 = 1, WOLFSSL_TLSV1_1 = 2 or WOLFSSL_TLSV1_2 = 3.

See: SetMinVersionHelper

Return:

- SSL_SUCCESS returned if this function and its subroutine executes without error.
- BAD_FUNC_ARG returned if the SSL object is NULL. In the subroutine this error is thrown if there is not a good version match.

Example

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(protocol method);
WOLFSSL* ssl = WOLFSSL_new(ctx);
int version;  macro representation
…
if(wolfSSL_CTX_SetMinVersion(ssl->ctx, version) != SSL_SUCCESS){
    Failed to set min version
}
```

C.58.4.286 function wolfSSL_GetObjectSize

```
int wolfSSL_GetObjectSize(
    void
)
```

This function returns the size of the WOLFSSL object and will be dependent on build options and settings. If SHOW_SIZES has been defined when building wolfSSL, this function will also print the sizes of individual objects within the WOLFSSL object (Suites, Ciphers, etc.) to stdout.

Parameters:

- **none** No parameters.

See: wolfSSL_new

Return: size This function returns the size of the WOLFSSL object.

Example

```
int size = 0;
size = wolfSSL_GetObjectSize();
printf("sizeof(WOLFSSL) = %d\n", size);
```

C.58.4.287 function wolfSSL_GetOutputSize

```
int wolfSSL_GetOutputSize(
    WOLFSSL * ssl,
    int inSz
)
```

Returns the record layer size of the plaintext input. This is helpful when an application wants to know how many bytes will be sent across the Transport layer, given a specified plaintext input size. This function must be called after the SSL/TLS handshake has been completed.

Parameters:

- **ssl** a pointer to a WOLFSSL object, created using wolfSSL_new().
- **inSz** size of plaintext data.

See: wolfSSL_GetMaxOutputSize

Return:

- size Upon success, the requested size will be returned
- INPUT_SIZE_E will be returned if the input size is greater than the maximum TLS fragment size (see wolfSSL_GetMaxOutputSize())
- BAD_FUNC_ARG will be returned upon invalid function argument, or if the SSL/TLS handshake has not been completed yet

Example

none

C.58.4.288 function wolfSSL_GetMaxOutputSize

```
int wolfSSL_GetMaxOutputSize(
    WOLFSSL * ssl
)
```

Returns the maximum record layer size for plaintext data. This will correspond to either the maximum SSL/TLS record size as specified by the protocol standard, the maximum TLS fragment size as set by the TLS Max Fragment Length extension. This function is helpful when the application has called wolfSSL_GetOutputSize() and received a INPUT_SIZE_E error. This function must be called after the SSL/TLS handshake has been completed.

Parameters:

- **ssl** a pointer to a WOLFSSL object, created using wolfSSL_new().

See: wolfSSL_GetOutputSize

Return:

- size Upon success, the maximum output size will be returned
- BAD_FUNC_ARG will be returned upon invalid function argument, or if the SSL/TLS handshake has not been completed yet.

Example

none

C.58.4.289 function wolfSSL_SetVersion

```
int wolfSSL_SetVersion(
    WOLFSSL * ssl,
    int version
)
```

This function sets the SSL/TLS protocol version for the specified SSL session (WOLFSSL object) using the version as specified by version. This will override the protocol setting for the SSL session (ssl) - originally defined and set by the SSL context (wolfSSL_CTX_new()) method type.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **version** SSL/TLS protocol version. Possible values include WOLFSSL_SSLV3, WOLFSSL_TLSV1, WOLFSSL_TLSV1_1, WOLFSSL_TLSV1_2.

See: wolfSSL_CTX_new

Return:

- SSL_SUCCESS upon success.
- BAD_FUNC_ARG will be returned if the input SSL object is NULL or an incorrect protocol version is given for version.

Example

```
int ret = 0;
WOLFSSL* ssl;
...

ret = wolfSSL_SetVersion(ssl, WOLFSSL_TLSV1);
if (ret != SSL_SUCCESS) {
    // failed to set SSL session protocol version
}
```

C.58.4.290 function wolfSSL_CTX_SetMacEncryptCb

```
void wolfSSL_CTX_SetMacEncryptCb(
    WOLFSSL_CTX * ctx,
    CallbackMacEncrypt cb
)
```

Allows caller to set the Atomic User Record Processing Mac/Encrypt Callback. The callback should return 0 for success or < 0 for an error. The ssl and ctx pointers are available for the user's convenience. macOut is the output buffer where the result of the mac should be stored. macIn is the mac input buffer and macInSz notes the size of the buffer. macContent and macVerify are needed for wolfSSL_SetTlsHmacInner() and be passed along as is. encOut is the output buffer where the result on the encryption should be stored. encIn is the input buffer to encrypt while encSz is the size of the input. An example callback can be found wolfssl/test.h myMacEncryptCb().

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **cb** callback function to register for Mac/Encrypt.

See:

- wolfSSL_SetMacEncryptCtx
- wolfSSL_GetMacEncryptCtx

Return: none No return.

Example

none

C.58.4.291 function wolfSSL_SetMacEncryptCtx

```
void wolfSSL_SetMacEncryptCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

Allows caller to set the Atomic User Record Processing Mac/Encrypt Callback Context to ctx.

Parameters:

- **ssl** pointer to a WOLFSSL structure, created using wolfSSL_new().
- **ctx** pointer to the user context to be stored.

See:

- wolfSSL_CTX_SetMacEncryptCb
- wolfSSL_GetMacEncryptCtx

Return: none No return.

Example

none

C.58.4.292 function wolfSSL_GetMacEncryptCtx

```
void * wolfSSL_GetMacEncryptCtx(
    WOLFSSL * ssl
)
```

Allows caller to retrieve the Atomic User Record Processing Mac/Encrypt Callback Context previously stored with wolfSSL_SetMacEncryptCtx().

Parameters:

- **ssl** pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_CTX_SetMacEncryptCb
- wolfSSL_SetMacEncryptCtx

Return:

- pointer If successful the call will return a valid pointer to the context.
- NULL will be returned for a blank context.

Example

none

C.58.4.293 function wolfSSL_CTX_SetDecryptVerifyCb

```
void wolfSSL_CTX_SetDecryptVerifyCb(
    WOLFSSL_CTX * ctx,
    CallbackDecryptVerify cb
)
```

Allows caller to set the Atomic User Record Processing Decrypt/Verify Callback. The callback should return 0 for success or < 0 for an error. The ssl and ctx pointers are available for the user's convenience. decOut is the output buffer where the result of the decryption should be stored. decIn is the encrypted input buffer and decInSz notes the size of the buffer. content and verify are needed for wolfSSL_SetTlsHmacInner() and be passed along as is. padSz is an output variable that should be set with the total value of the padding. That is, the mac size plus any padding and pad bytes. An example callback can be found wolfssl/test.h myDecryptVerifyCb().

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **cb** callback function to register for Decrypt/Verify.

See:

- wolfSSL_SetMacEncryptCtx
- wolfSSL_GetMacEncryptCtx

Return: none No returns.

Example

none

C.58.4.294 function wolfSSL_SetDecryptVerifyCtx

```
void wolfSSL_SetDecryptVerifyCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

Allows caller to set the Atomic User Record Processing Decrypt/Verify Callback Context to ctx.

Parameters:

- **ssl** pointer to a WOLFSSL structure, created using wolfSSL_new().
- **ctx** pointer to the user context to be stored.

See:

- wolfSSL_CTX_SetDecryptVerifyCb
- wolfSSL_GetDecryptVerifyCtx

Return: none No returns.

Example

none

C.58.4.295 function wolfSSL_GetDecryptVerifyCtx

```
void * wolfSSL_GetDecryptVerifyCtx(
    WOLFSSL * ssl
)
```

Allows caller to retrieve the Atomic User Record Processing Decrypt/Verify Callback Context previously stored with wolfSSL_SetDecryptVerifyCtx().

Parameters:

- **ssl** pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_CTX_SetDecryptVerifyCb
- wolfSSL_SetDecryptVerifyCtx

Return:

- pointer If successful the call will return a valid pointer to the context.
- NULL will be returned for a blank context.

Example

none

C.58.4.296 function wolfSSL_GetMacSecret

```
const unsigned char * wolfSSL_GetMacSecret(
    WOLFSSL * ssl,
    int verify
)
```

Allows retrieval of the Hmac/Mac secret from the handshake process. The verify parameter specifies whether this is for verification of a peer message.

Parameters:

- **ssl** a pointer to a WOLFSSL object, created using wolfSSL_new().
- **verify** specifies whether this is for verification of a peer message.

See: wolfSSL_GetHmacSize

Return:

- pointer If successful the call will return a valid pointer to the secret. The size of the secret can be obtained from wolfSSL_GetHmacSize().
- NULL will be returned for an error state.

Example

none

C.58.4.297 function wolfSSL_GetClientWriteKey

```
const unsigned char * wolfSSL_GetClientWriteKey(
    WOLFSSL *
)
```

Allows retrieval of the client write key from the handshake process.

Parameters:

- **ssl** a pointer to a WOLFSSL object, created using wolfSSL_new().

See:

- wolfSSL_GetKeySize
- wolfSSL_GetClientWriteIV

Return:

- pointer If successful the call will return a valid pointer to the key. The size of the key can be obtained from wolfSSL_GetKeySize().
- NULL will be returned for an error state.

Example

none

C.58.4.298 function wolfSSL_GetClientWriteIV

```
const unsigned char * wolfSSL_GetClientWriteIV(
    WOLFSSL *
)
```

Allows retrieval of the client write IV (initialization vector) from the handshake process.

Parameters:

- **ssl** a pointer to a WOLFSSL object, created using wolfSSL_new().

See:

- wolfSSL_GetCipherBlockSize()
- wolfSSL_GetClientWriteKey()

Return:

- pointer If successful the call will return a valid pointer to the IV. The size of the IV can be obtained from wolfSSL_GetCipherBlockSize().
- NULL will be returned for an error state.

Example

none

C.58.4.299 function wolfSSL_GetServerWriteKey

```
const unsigned char * wolfSSL_GetServerWriteKey(
    WOLFSSL *
)
```

Allows retrieval of the server write key from the handshake process.

Parameters:

- **ssl** a pointer to a WOLFSSL object, created using wolfSSL_new().

See:

- wolfSSL_GetKeySize
- wolfSSL_GetServerWriteIV

Return:

- pointer If successful the call will return a valid pointer to the key. The size of the key can be obtained from wolfSSL_GetKeySize().
- NULL will be returned for an error state.

Example

none

C.58.4.300 function wolfSSL_GetServerWriteIV

```
const unsigned char * wolfSSL_GetServerWriteIV(
    WOLFSSL *
)
```

Allows retrieval of the server write IV (initialization vector) from the handshake process.

Parameters:

- **ssl** a pointer to a WOLFSSL object, created using wolfSSL_new().

See:

- wolfSSL_GetCipherBlockSize
- wolfSSL_GetClientWriteKey

Return:

- pointer If successful the call will return a valid pointer to the IV. The size of the IV can be obtained from wolfSSL_GetCipherBlockSize().
- NULL will be returned for an error state.

C.58.4.301 function wolfSSL_GetKeySize

```
int wolfSSL_GetKeySize(
    WOLFSSL * ssl
)
```

Allows retrieval of the key size from the handshake process.

Parameters:

- **ssl** a pointer to a WOLFSSL object, created using wolfSSL_new().

See:

- wolfSSL_GetClientWriteKey
- wolfSSL_GetServerWriteKey

Return:

- size If successful the call will return the key size in bytes.
- BAD_FUNC_ARG will be returned for an error state.

Example

none

C.58.4.302 function wolfSSL_GetIVSize

```
int wolfSSL_GetIVSize(
    WOLFSSL * ssl
)
```

Returns the iv_size member of the specs structure held in the WOLFSSL struct.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_GetKeySize
- wolfSSL_GetClientWriteIV
- wolfSSL_GetServerWriteIV

Return:

- iv_size returns the value held in ssl->specs.iv_size.
- BAD_FUNC_ARG returned if the WOLFSSL structure is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
int ivSize;
...
ivSize = wolfSSL_GetIVSize(ssl);

if(ivSize > 0){
    // ivSize holds the specs.iv_size value.
}
```

C.58.4.303 function wolfSSL_GetSide

```
int wolfSSL_GetSide(
    WOLFSSL * ssl
)
```

Allows retrieval of the side of this WOLFSSL connection.

Parameters:

- **ssl** a pointer to a WOLFSSL object, created using wolfSSL_new().

See:

- wolfSSL_GetClientWriteKey
- wolfSSL_GetServerWriteKey

Return:

- success If successful the call will return either WOLFSSL_SERVER_END or WOLFSSL_CLIENT_END depending on the side of WOLFSSL object.
- BAD_FUNC_ARG will be returned for an error state.

Example

none

C.58.4.304 function wolfSSL_IsTLSv1_1

```
int wolfSSL_IsTLSv1_1(
    WOLFSSL * ssl
)
```

Allows caller to determine if the negotiated protocol version is at least TLS version 1.1 or greater.

Parameters:

- **ssl** a pointer to a WOLFSSL object, created using wolfSSL_new().

See: wolfSSL_GetSide

Return:

- true/false If successful the call will return 1 for true or 0 for false.
- BAD_FUNC_ARG will be returned for an error state.

Example

none

C.58.4.305 function wolfSSL_GetBulkCipher

```
int wolfSSL_GetBulkCipher(
    WOLFSSL * ssl
)
```

Allows caller to determine the negotiated bulk cipher algorithm from the handshake.

Parameters:

- **ssl** a pointer to a WOLFSSL object, created using wolfSSL_new().

See:

- wolfSSL_GetCipherBlockSize
- wolfSSL_GetKeySize

Return:

- If successful the call will return one of the following: wolfssl_cipher_null, wolfssl_des, wolfssl_triple_des, wolfssl_aes, wolfssl_aes_gcm, wolfssl_aes_ccm, wolfssl_camellia.
- BAD_FUNC_ARG will be returned for an error state.

Example

none

C.58.4.306 function wolfSSL_GetCipherBlockSize

```
int wolfSSL_GetCipherBlockSize(
    WOLFSSL * ssl
)
```

Allows caller to determine the negotiated cipher block size from the handshake.

Parameters:

- **ssl** a pointer to a WOLFSSL object, created using wolfSSL_new().

See:

- wolfSSL_GetBulkCipher
- wolfSSL_GetKeySize

Return:

- size If successful the call will return the size in bytes of the cipher block size.
- BAD_FUNC_ARG will be returned for an error state.

Example

none

C.58.4.307 function wolfSSL_GetAeadMacSize

```
int wolfSSL_GetAeadMacSize(
    WOLFSSL * ssl
)
```

Allows caller to determine the negotiated aead mac size from the handshake. For cipher type WOLFSSL_AEAD_TYPE.

Parameters:

- **ssl** a pointer to a WOLFSSL object, created using wolfSSL_new().

See:

- wolfSSL_GetBulkCipher
- wolfSSL_GetKeySize

Return:

- size If successful the call will return the size in bytes of the aead mac size.
- BAD_FUNC_ARG will be returned for an error state.

Example

none

C.58.4.308 function wolfSSL_GetHmacSize

```
int wolfSSL_GetHmacSize(
    WOLFSSL * ssl
)
```

Allows caller to determine the negotiated (h)mac size from the handshake. For cipher types except WOLFSSL_AEAD_TYPE.

Parameters:

- **ssl** a pointer to a WOLFSSL object, created using wolfSSL_new().

See:

- wolfSSL_GetBulkCipher
- wolfSSL_GetHmacType

Return:

- size If successful the call will return the size in bytes of the (h)mac size.
- BAD_FUNC_ARG will be returned for an error state.

Example

none

C.58.4.309 function wolfSSL_GetHmacType

```
int wolfSSL_GetHmacType(
    WOLFSSL * ssl
)
```

Allows caller to determine the negotiated (h)mac type from the handshake. For cipher types except WOLFSSL_AEAD_TYPE.

Parameters:

- **ssl** a pointer to a WOLFSSL object, created using wolfSSL_new().

See:

- wolfSSL_GetBulkCipher
- wolfSSL_GetHmacSize

Return:

- If successful the call will return one of the following: MD5, SHA, SHA256, SHA384.
- BAD_FUNC_ARG may be returned for an error state.
- SSL_FATAL_ERROR may also be returned for an error state.

Example

none

C.58.4.310 function wolfSSL_GetCipherType

```
int wolfSSL_GetCipherType(
    WOLFSSL * ssl
)
```

Allows caller to determine the negotiated cipher type from the handshake.

Parameters:

- **ssl** a pointer to a WOLFSSL object, created using wolfSSL_new().

See:

- wolfSSL_GetBulkCipher
- wolfSSL_GetHmacType

Return:

- If successful the call will return one of the following: WOLFSSL_BLOCK_TYPE, WOLFSSL_STREAM_TYPE, WOLFSSL_AEAD_TYPE.
- BAD_FUNC_ARG will be returned for an error state.

Example

none

C.58.4.311 function wolfSSL_SetTlsHmacInner

```
int wolfSSL_SetTlsHmacInner(
    WOLFSSL * ssl,
    byte * inner,
    word32 sz,
    int content,
    int verify
)
```

Allows caller to set the Hmac Inner vector for message sending/receiving. The result is written to inner which should be at least wolfSSL_GetHmacSize() bytes. The size of the message is specified by sz, content is the type of message, and verify specifies whether this is a verification of a peer message. Valid for cipher types excluding WOLFSSL_AEAD_TYPE.

Parameters:

- **none** No parameters.

See:

- wolfSSL_GetBulkCipher
- wolfSSL_GetHmacType

Return:

- 1 upon success.
- BAD_FUNC_ARG will be returned for an error state.

Example

none

C.58.4.312 function wolfSSL_CTX_SetEccSignCb

```
void wolfSSL_CTX_SetEccSignCb(
    WOLFSSL_CTX * ctx,
    CallbackEccSign cb
)
```

Allows caller to set the Public Key Callback for ECC Signing. The callback should return 0 for success or < 0 for an error. The ssl and ctx pointers are available for the user's convenience. in is the input buffer to sign while inSz denotes the length of the input. out is the output buffer where the result of the signature should be stored. outSz is an input/output variable that specifies the size of the output buffer upon invocation and the actual size of the signature should be stored there before returning. keyDer is the ECC Private key in ASN1 format and keySz is the length of the key in bytes. An example callback can be found wolfssl/test.h myEccSign().

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **cb** callback function to register for ECC signing.

See:

- wolfSSL_SetEccSignCtx
- wolfSSL_GetEccSignCtx

Return: none No returns.

Example

none

C.58.4.313 function wolfSSL_SetEccSignCtx

```
void wolfSSL_SetEccSignCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

Allows caller to set the Public Key Ecc Signing Callback Context to ctx.

Parameters:

- **ssl** a pointer to a WOLFSSL object, created using wolfSSL_new().
- **ctx** a pointer to the user context to be stored

See:

- wolfSSL_CTX_SetEccSignCb
- wolfSSL_GetEccSignCtx

Return: none No returns.

Example

none

C.58.4.314 function wolfSSL_GetEccSignCtx

```
void * wolfSSL_GetEccSignCtx(
    WOLFSSL * ssl
)
```

Allows caller to retrieve the Public Key Ecc Signing Callback Context previously stored with wolfSSL_SetEccSignCtx().

Parameters:

- **ssl** a pointer to a WOLFSSL object, created using wolfSSL_new().

See:

- wolfSSL_CTX_SetEccSignCb
- wolfSSL_SetEccSignCtx

Return:

- pointer If successful the call will return a valid pointer to the context.
- NULL will be returned for a blank context.

Example

none

C.58.4.315 function wolfSSL_CTX_SetEccSignCtx

```
void wolfSSL_CTX_SetEccSignCtx(
    WOLFSSL_CTX * ctx,
    void * userCtx
)
```

Allows caller to set the Public Key Ecc Signing Callback Context to ctx.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().
- **ctx** a pointer to the user context to be stored

See:

- wolfSSL_CTX_SetEccSignCb
- wolfSSL_CTX_GetEccSignCtx

Return: none No returns.

Example

none

C.58.4.316 function wolfSSL_CTX_GetEccSignCtx

```
void * wolfSSL_CTX_GetEccSignCtx(
    WOLFSSL_CTX * ctx
)
```

Allows caller to retrieve the Public Key Ecc Signing Callback Context previously stored with wolfSSL_SetEccSignCtx().

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().

See:

- wolfSSL_CTX_SetEccSignCb
- wolfSSL_CTX_SetEccSignCtx

Return:

- pointer If successful the call will return a valid pointer to the context.
- NULL will be returned for a blank context.

Example

none

C.58.4.317 function wolfSSL_CTX_SetEccVerifyCb

```
void wolfSSL_CTX_SetEccVerifyCb(
    WOLFSSL_CTX * ctx,
    CallbackEccVerify cb
)
```

Allows caller to set the Public Key Callback for ECC Verification. The callback should return 0 for success or < 0 for an error. The ssl and ctx pointers are available for the user's convenience. sig is the signature to verify and sigSz denotes the length of the signature. hash is an input buffer containing the digest of the message and hashSz denotes the length in bytes of the hash. result is an output variable where the result of the verification should be stored, 1 for success and 0 for failure. keyDer is the ECC Private key in ASN1 format and keySz is the length of the key in bytes. keyDer is NULL with keySz 0 when only the callback holds the key, so the callback must then supply it. An example callback can be found wolfssl/test.h myEccVerify().

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().

- **cb** callback function to register for ECC verification.

See:

- wolfSSL_SetEccVerifyCtx
- wolfSSL_GetEccVerifyCtx

Return: none No returns.

Example

none

C.58.4.318 function wolfSSL_SetEccVerifyCtx

```
void wolfSSL_SetEccVerifyCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

Allows caller to set the Public Key Ecc Verification Callback Context to ctx.

Parameters:

- **ssl** pointer to a WOLFSSL structure, created using wolfSSL_new().
- **ctx** pointer to the user context to be stored.

See:

- wolfSSL_CTX_SetEccVerifyCb
- wolfSSL_GetEccVerifyCtx

Return: none No returns.

Example

none

C.58.4.319 function wolfSSL_GetEccVerifyCtx

```
void * wolfSSL_GetEccVerifyCtx(
    WOLFSSL * ssl
)
```

Allows caller to retrieve the Public Key Ecc Verification Callback Context previously stored with wolfSSL_SetEccVerifyCtx().

Parameters:

- **ssl** pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_CTX_SetEccVerifyCb
- wolfSSL_SetEccVerifyCtx

Return:

- pointer If successful the call will return a valid pointer to the context.
- NULL will be returned for a blank context.

Example

none

C.58.4.320 function wolfSSL_CTX_SetRsaSignCb

```
void wolfSSL_CTX_SetRsaSignCb(
    WOLFSSL_CTX * ctx,
    CallbackRsaSign cb
)
```

Allows caller to set the Public Key Callback for RSA Signing. The callback should return 0 for success or < 0 for an error. The ssl and ctx pointers are available for the user's convenience. in is the input buffer to sign while inSz denotes the length of the input. out is the output buffer where the result of the signature should be stored. outSz is an input/output variable that specifies the size of the output buffer upon invocation and the actual size of the signature should be stored there before returning. keyDer is the RSA Private key in ASN1 format and keySz is the length of the key in bytes. An example callback can be found wolfssl/test.h myRsaSign().

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **cb** callback function to register for RSA signing.

See:

- wolfSSL_SetRsaSignCtx
- wolfSSL_GetRsaSignCtx

Return: none No returns.

Example

none

C.58.4.321 function wolfSSL_SetRsaSignCtx

```
void wolfSSL_SetRsaSignCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

Allows caller to set the Public Key RSA Signing Callback Context to ctx.

Parameters:

- **ssl** pointer to a WOLFSSL structure, created using wolfSSL_new().
- **ctx** pointer to the user context to be stored.

See:

- wolfSSL_CTX_SetRsaSignCb
- wolfSSL_GetRsaSignCtx

Return: none No Returns.

Example

none

C.58.4.322 function wolfSSL_GetRsaSignCtx

```
void * wolfSSL_GetRsaSignCtx(
    WOLFSSL * ssl
)
```

Allows caller to retrieve the Public Key RSA Signing Callback Context previously stored with wolfSSL_SetRsaSignCtx().

Parameters:

- **ssl** pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_CTX_SetRsaSignCb
- wolfSSL_SetRsaSignCtx

Return:

- pointer If successful the call will return a valid pointer to the context.
- NULL will be returned for a blank context.

Example

none

C.58.4.323 function wolfSSL_CTX_SetRsaVerifyCb

```
void wolfSSL_CTX_SetRsaVerifyCb(
    WOLFSSL_CTX * ctx,
    CallbackRsaVerify cb
)
```

Allows caller to set the Public Key Callback for RSA Verification. The callback should return the number of plaintext bytes for success or < 0 for an error. The ssl and ctx pointers are available for the user's convenience. sig is the signature to verify and sigSz denotes the length of the signature. out should be set to the beginning of the verification buffer after the decryption process and any padding. keyDer is the RSA Public key in ASN1 format and keySz is the length of the key in bytes. An example callback can be found wolfssl/test.h myRsaVerify().

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **cb** callback function to register for RSA verification.

See:

- wolfSSL_SetRsaVerifyCtx
- wolfSSL_GetRsaVerifyCtx

Return: none No returns.

C.58.4.324 function wolfSSL_SetRsaVerifyCtx

```
void wolfSSL_SetRsaVerifyCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

Allows caller to set the Public Key RSA Verification Callback Context to ctx.

Parameters:

- **ssl** pointer to a WOLFSSL structure, created using wolfSSL_new().
- **ctx** pointer to the user context to be stored.

See:

- wolfSSL_CTX_SetRsaVerifyCb

- wolfSSL_GetRsaVerifyCtx

Return: none No returns.

Example

none

C.58.4.325 function wolfSSL_GetRsaVerifyCtx

```
void * wolfSSL_GetRsaVerifyCtx(
    WOLFSSL * ssl
)
```

Allows caller to retrieve the Public Key RSA Verification Callback Context previously stored with wolfSSL_SetRsaVerifyCtx().

Parameters:

- **ssl** pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_CTX_SetRsaVerifyCb
- wolfSSL_SetRsaVerifyCtx

Return:

- pointer If successful the call will return a valid pointer to the context.
- NULL will be returned for a blank context.

Example

none

C.58.4.326 function wolfSSL_CTX_SetRsaEncCb

```
void wolfSSL_CTX_SetRsaEncCb(
    WOLFSSL_CTX * ctx,
    CallbackRsaEnc cb
)
```

Allows caller to set the Public Key Callback for RSA Public Encrypt. The callback should return 0 for success or < 0 for an error. The ssl and ctx pointers are available for the user's convenience. in is the input buffer to encrypt while inSz denotes the length of the input. out is the output buffer where the result of the encryption should be stored. outSz is an input/output variable that specifies the size of the output buffer upon invocation and the actual size of the encryption should be stored there before returning. keyDer is the RSA Public key in ASN1 format and keySz is the length of the key in bytes. An example callback can be found wolfssl/test.h myRsaEnc().

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **cb** callback function to register for RSA public encrypt.

See:

- wolfSSL_SetRsaEncCtx
- wolfSSL_GetRsaEncCtx

Return: none No returns.

Examples

none

C.58.4.327 function wolfSSL_SetRsaEncCtx

```
void wolfSSL_SetRsaEncCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

Allows caller to set the Public Key RSA Public Encrypt Callback Context to ctx.

Parameters:

- **ssl** pointer to a WOLFSSL structure, created using wolfSSL_new().
- **ctx** pointer to the user context to be stored.

See:

- wolfSSL_CTX_SetRsaEncCb
- wolfSSL_GetRsaEncCtx

Return: none No returns.

Example

none

C.58.4.328 function wolfSSL_GetRsaEncCtx

```
void * wolfSSL_GetRsaEncCtx(
    WOLFSSL * ssl
)
```

Allows caller to retrieve the Public Key RSA Public Encrypt Callback Context previously stored with wolfSSL_SetRsaEncCtx().

Parameters:

- **ssl** pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_CTX_SetRsaEncCb
- wolfSSL_SetRsaEncCtx

Return:

- pointer If successful the call will return a valid pointer to the context.
- NULL will be returned for a blank context.

Example

none

C.58.4.329 function wolfSSL_CTX_SetRsaDecCb

```
void wolfSSL_CTX_SetRsaDecCb(
    WOLFSSL_CTX * ctx,
    CallbackRsaDec cb
)
```

Allows caller to set the Public Key Callback for RSA Private Decrypt. The callback should return the number of plaintext bytes for success or < 0 for an error. The ssl and ctx pointers are available for the user's convenience. in is the input buffer to decrypt and inSz denotes the length of the input. out should be set to the beginning of the decryption buffer after the decryption process and any padding. keyDer is the RSA Private key in ASN1 format and keySz is the length of the key in bytes. An example callback can be found wolfssl/test.h myRsaDec().

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **cb** callback function to register for RSA private decrypt.

See:

- wolfSSL_SetRsaDecCtx
- wolfSSL_GetRsaDecCtx

Return: none No returns.

Example

none

C.58.4.330 function wolfSSL_SetRsaDecCtx

```
void wolfSSL_SetRsaDecCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

Allows caller to set the Public Key RSA Private Decrypt Callback Context to ctx.

Parameters:

- **ssl** pointer to a WOLFSSL structure, created using wolfSSL_new().
- **ctx** pointer to the user context to be stored.

See:

- wolfSSL_CTX_SetRsaDecCb
- wolfSSL_GetRsaDecCtx

Return: none No returns.

Example

none

C.58.4.331 function wolfSSL_GetRsaDecCtx

```
void * wolfSSL_GetRsaDecCtx(
    WOLFSSL * ssl
)
```

Allows caller to retrieve the Public Key RSA Private Decrypt Callback Context previously stored with wolfSSL_SetRsaDecCtx().

Parameters:

- **ssl** pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_CTX_SetRsaDecCb

- wolfSSL_SetRsaDecCtx

Return:

- pointer If successful the call will return a valid pointer to the context.
- NULL will be returned for a blank context.

Example

none

C.58.4.332 function wolfSSL_CTX_SetCACb

```
void wolfSSL_CTX_SetCACb(
    WOLFSSL_CTX * ctx,
    CallbackCACache cb
)
```

This function registers a callback with the SSL context (WOLFSSL_CTX) to be called when a new CA certificate is loaded into wolfSSL. The callback is given a buffer with the DER-encoded certificate.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **cb** function to be registered as the CA callback for the wolfSSL context, ctx. The signature of this function must follow that as shown above in the Synopsis section.

See: wolfSSL_CTX_load_verify_locations

Return: none No return.

Example

```
WOLFSSL_CTX* ctx = 0;

// CA callback prototype
int MyCACallback(unsigned char *der, int sz, int type);

// Register the custom CA callback with the SSL context
wolfSSL_CTX_SetCACb(ctx, MyCACallback);

int MyCACallback(unsigned char* der, int sz, int type)
{
    // custom CA callback function, DER-encoded cert
    // located in “der” of size “sz” with type “type”
}
```

C.58.4.333 function wolfSSL_CertManagerNew_ex

```
WOLFSSL_CERT_MANAGER * wolfSSL_CertManagerNew_ex(
    void * heap
)
```

Allocates and initializes a new Certificate Manager context. This context may be used independent of SSL needs. It may be used to load certificates, verify certificates, and check the revocation status.

Parameters:

- **heap** pointer to a heap hint for memory allocation.

See: wolfSSL_CertManagerFree

Return:

- WOLFSSL_CERT_MANAGER If successful the call will return a valid WOLFSSL_CERT_MANAGER pointer.
- NULL will be returned for an error state.

C.58.4.334 function wolfSSL_CertManagerNew

```
WOLFSSL_CERT_MANAGER * wolfSSL_CertManagerNew(
    void
)
```

Allocates and initializes a new Certificate Manager context. This context may be used independent of SSL needs. It may be used to load certificates, verify certificates, and check the revocation status.

Parameters:

- **none** No parameters.

See: wolfSSL_CertManagerFree

Return:

- WOLFSSL_CERT_MANAGER If successful the call will return a valid WOLFSSL_CERT_MANAGER pointer.
- NULL will be returned for an error state.

Example

```
#import <wolfssl/ssl.h>

WOLFSSL_CERT_MANAGER* cm;
cm = wolfSSL_CertManagerNew();
if (cm == NULL) {
// error creating new cert manager
}
```

C.58.4.335 function wolfSSL_CertManagerFree

```
void wolfSSL_CertManagerFree(
    WOLFSSL_CERT_MANAGER * cm
)
```

Frees all resources associated with the Certificate Manager context. Call this when you no longer need to use the Certificate Manager.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().

See: wolfSSL_CertManagerNew

Return: none

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CERT_MANAGER* cm;
...
wolfSSL_CertManagerFree(cm);
```

C.58.4.336 function wolfSSL_CertManagerLoadCA

```
int wolfSSL_CertManagerLoadCA(
    WOLFSSL_CERT_MANAGER * cm,
    const char * f,
    const char * d
)
```

Specifies the locations for CA certificate loading into the manager context. The PEM certificate CAfile may contain several trusted CA certificates. If CApath is not NULL it specifies a directory containing CA certificates in PEM format.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().
- **file** pointer to the name of the file containing CA certificates to load.
- **path** pointer to the name of a directory path containing CA c ertificates to load. The NULL pointer may be used if no certificate directory is desired.

See: wolfSSL_CertManagerVerify

Return:

- SSL_SUCCESS If successful the call will return.
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- BAD_FUNC_ARG is the error that will be returned if a pointer is not provided.
- SSL_FATAL_ERROR - will be returned upon failure.

Example

```
#include <wolfssl/ssl.h>

int ret = 0;
WOLFSSL_CERT_MANAGER* cm;
...
ret = wolfSSL_CertManagerLoadCA(cm, “path/to/cert-file.pem”, 0);
if (ret != SSL_SUCCESS) {
// error loading CA certs into cert manager
}
```

C.58.4.337 function wolfSSL_CertManagerLoadCABuffer

```
int wolfSSL_CertManagerLoadCABuffer(
    WOLFSSL_CERT_MANAGER * cm,
    const unsigned char * buff,
    long sz,
    int format
)
```

Loads the CA Buffer by calling wolfSSL_CTX_load_verify_buffer and returning that result using a temporary cm so as not to lose the information in the cm passed into the function.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().
- **in** buffer for cert information.
- **sz** length of the buffer.

- **format** certificate format, either PEM or DER.

See:

- wolfSSL_CTX_load_verify_buffer
- ProcessChainBuffer
- ProcessBuffer
- cm_pick_method

Return:

- SSL_FATAL_ERROR is returned if the WOLFSSL_CERT_MANAGER struct is NULL or if wolfSSL_CTX_new() returns NULL.
- SSL_SUCCESS is returned for a successful execution.

Example

```
WOLFSSL_CERT_MANAGER* cm = (WOLFSSL_CERT_MANAGER*)vp;
…
const unsigned char* in;
long sz;
int format;
…
if(wolfSSL_CertManagerLoadCABuffer(vp, sz, format) != SSL_SUCCESS){
    Error returned. Failure case code block.
}
```

C.58.4.338 function wolfSSL_CertManagerUnloadCAs

```
int wolfSSL_CertManagerUnloadCAs(
    WOLFSSL_CERT_MANAGER * cm
)
```

This function unloads the CA signer list.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().

See: UnlockMutex

Return:

- SSL_SUCCESS returned on successful execution of the function.
- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER is NULL.
- BAD_MUTEX_E returned if there was a mutex error.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = wolfSSL_CTX_new(protocol method);
WOLFSSL_CERT_MANAGER* cm = wolfSSL_CTX_GetCertManager(ctx);
...
if(wolfSSL_CertManagerUnloadCAs(cm) != SSL_SUCCESS){
    Failure case.
}
```

C.58.4.339 function wolfSSL_CertManagerUnloadIntermediateCerts

```
int wolfSSL_CertManagerUnloadIntermediateCerts(
    WOLFSSL_CERT_MANAGER * cm
)
```

This function unloads intermediate certificates add to the CA signer list.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().

See: UnlockMutex

Return:

- SSL_SUCCESS returned on successful execution of the function.
- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER is NULL.
- BAD_MUTEX_E returned if there was a mutex error.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = wolfSSL_CTX_new(protocol method);
WOLFSSL_CERT_MANAGER* cm = wolfSSL_CTX_GetCertManager(ctx);
...
if(wolfSSL_CertManagerUnloadIntermediateCerts(cm) != SSL_SUCCESS){
    Failure case.
}
```

C.58.4.340 function wolfSSL_CertManagerUnload_trust_peers

```
int wolfSSL_CertManagerUnload_trust_peers(
    WOLFSSL_CERT_MANAGER * cm
)
```

The function will free the Trusted Peer linked list and unlocks the trusted peer list.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().

See: UnLockMutex

Return:

- SSL_SUCCESS if the function completed normally.
- BAD_FUNC_ARG if the WOLFSSL_CERT_MANAGER is NULL.
- BAD_MUTEX_E mutex error if tpLock, a member of the WOLFSSL_CERT_MANAGER struct, is 0 (nill).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(Protocol define);
WOLFSSL_CERT_MANAGER* cm = wolfSSL_CertManagerNew();
...
if(wolfSSL_CertManagerUnload_trust_peers(cm) != SSL_SUCCESS){
    The function did not execute successfully.
}
```

C.58.4.341 function wolfSSL_CertManagerVerify

```
int wolfSSL_CertManagerVerify(
    WOLFSSL_CERT_MANAGER * cm,
    const char * f,
    int format
)
```

Specifies the certificate to verify with the Certificate Manager context. The format can be SSL_FILETYPE_PEM or SSL_FILETYPE_ASN1.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().
- **fname** pointer to the name of the file containing the certificates to verify.
- **format** format of the certificate to verify - either SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CertManagerLoadCA
- wolfSSL_CertManagerVerifyBuffer

Return:

- SSL_SUCCESS If successful.
- ASN_SIG_CONFIRM_E will be returned if the signature could not be verified.
- ASN_SIG_OID_E will be returned if the signature type is not supported.
- CRL_CERT_REVOKED is an error that is returned if this certificate has been revoked.
- CRL_MISSING is an error that is returned if a current issuer CRL is not available.
- ASN_BEFORE_DATE_E will be returned if the current date is before the before date.
- ASN_AFTER_DATE_E will be returned if the current date is after the after date.
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- BAD_FUNC_ARG is the error that will be returned if a pointer is not provided.

Example

```
int ret = 0;
WOLFSSL_CERT_MANAGER* cm;
...

ret = wolfSSL_CertManagerVerify(cm, “path/to/cert-file.pem”,
SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    error verifying certificate
}
```

C.58.4.342 function wolfSSL_CertManagerVerifyBuffer

```
int wolfSSL_CertManagerVerifyBuffer(
    WOLFSSL_CERT_MANAGER * cm,
    const unsigned char * buff,
    long sz,
    int format
)
```

Specifies the certificate buffer to verify with the Certificate Manager context. The format can be SSL_FILETYPE_PEM or SSL_FILETYPE_ASN1.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().
- **buff** buffer containing the certificates to verify.
- **sz** size of the buffer, buf.
- **format** format of the certificate to verify, located in buf - either SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CertManagerLoadCA
- wolfSSL_CertManagerVerify

Return:

- SSL_SUCCESS If successful.
- ASN_SIG_CONFIRM_E will be returned if the signature could not be verified.
- ASN_SIG_OID_E will be returned if the signature type is not supported.
- CRL_CERT_REVOKED is an error that is returned if this certificate has been revoked.
- CRL_MISSING is an error that is returned if a current issuer CRL is not available.
- ASN_BEFORE_DATE_E will be returned if the current date is before the before date.
- ASN_AFTER_DATE_E will be returned if the current date is after the after date.
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- BAD_FUNC_ARG is the error that will be returned if a pointer is not provided.

Example

```
#include <wolfssl/ssl.h>

int ret = 0;
int sz = 0;
WOLFSSL_CERT_MANAGER* cm;
byte certBuff[...];
...

ret = wolfSSL_CertManagerVerifyBuffer(cm, certBuff, sz, SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    error verifying certificate
}
```

C.58.4.343 function wolfSSL_CertManagerSetVerify

```
void wolfSSL_CertManagerSetVerify(
    WOLFSSL_CERT_MANAGER * cm,
    VerifyCallback verify_callback
)
```

The function sets the verifyCallback function in the Certificate Manager. If present, it will be called for each cert loaded. If there is a verification error, the verify callback can be used to over-ride the error.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().
- **verify_callback** a VerifyCallback function pointer to the callback routine

See: wolfSSL_CertManagerVerify

Return: none No return.

Example

```
#include <wolfssl/ssl.h>

int myVerify(int preverify, WOLFSSL_X509_STORE_CTX* store)
{ // do custom verification of certificate }

WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(Protocol define);
WOLFSSL_CERT_MANAGER* cm = wolfSSL_CertManagerNew();
...
wolfSSL_CertManagerSetVerify(cm, myVerify);
```

C.58.4.344 function wolfSSL_CertManagerCheckCRL

```
int wolfSSL_CertManagerCheckCRL(
    WOLFSSL_CERT_MANAGER * cm,
    const unsigned char * der,
    int sz
)
```

Check CRL if the option is enabled and compares the cert to the CRL list.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER struct.
- **der** pointer to a DER formatted certificate.
- **sz** size of the certificate.

See:

- CheckCertCRL
- ParseCertRelative
- wolfSSL_CertManagerSetCRL_CB
- InitDecodedCert

Return:

- SSL_SUCCESS returns if the function returned as expected. If the crlEnabled member of the WOLFSSL_CERT_MANAGER struct is turned on.
- MEMORY_E returns if the allocated memory failed.
- BAD_FUNC_ARG if the WOLFSSL_CERT_MANAGER is NULL.

Example

```
WOLFSSL_CERT_MANAGER* cm;
byte* der;
int sz; // size of der
...
if(wolfSSL_CertManagerCheckCRL(cm, der, sz) != SSL_SUCCESS){
    // Error returned. Deal with failure case.
}
```

C.58.4.345 function wolfSSL_CertManagerEnableCRL

```
int wolfSSL_CertManagerEnableCRL(
    WOLFSSL_CERT_MANAGER * cm,
    int options
)
```

Turns on Certificate Revocation List checking when verifying certificates with the Certificate Manager. By default, CRL checking is off. options include WOLFSSL_CRL_CHECKALL which performs CRL checking on each certificate in the chain versus the Leaf certificate only which is the default.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().
- **options** options to use when enabling the Certification Manager, cm.

See: wolfSSL_CertManagerDisableCRL

Return:

- SSL_SUCCESS If successful the call will return.
- NOT_COMPILED_IN will be returned if wolfSSL was not built with CRL enabled.
- MEMORY_E will be returned if an out of memory condition occurs.
- BAD_FUNC_ARG is the error that will be returned if a pointer is not provided.
- SSL_FAILURE will be returned if the CRL context cannot be initialized properly.

Example

```
#include <wolfssl/ssl.h>

int ret = 0;
WOLFSSL_CERT_MANAGER* cm;
...

ret = wolfSSL_CertManagerEnableCRL(cm, 0);
if (ret != SSL_SUCCESS) {
    error enabling cert manager
}

...
```

C.58.4.346 function wolfSSL_CertManagerDisableCRL

```
int wolfSSL_CertManagerDisableCRL(
    WOLFSSL_CERT_MANAGER * cm
)
```

Turns off Certificate Revocation List checking when verifying certificates with the Certificate Manager. By default, CRL checking is off. You can use this function to temporarily or permanently disable CRL checking with this Certificate Manager context that previously had CRL checking enabled.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().

See: wolfSSL_CertManagerEnableCRL

Return:

- SSL_SUCCESS If successful the call will return.
- BAD_FUNC_ARG is the error that will be returned if a function pointer is not provided.

Example

```
#include <wolfssl/ssl.h>

int ret = 0;
WOLFSSL_CERT_MANAGER* cm;
...
```

```
ret = wolfSSL_CertManagerDisableCRL(cm);
if (ret != SSL_SUCCESS) {
    error disabling cert manager
}
...
```

C.58.4.347 function wolfSSL_CertManagerLoadCRL

```
int wolfSSL_CertManagerLoadCRL(
    WOLFSSL_CERT_MANAGER * cm,
    const char * path,
    int type,
    int monitor
)
```

Error checks and passes through to LoadCRL() in order to load the cert into the CRL for revocation checking. An updated CRL can be loaded by first calling wolfSSL_CertManagerFreeCRL, then loading the new CRL.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().
- **path** a constant char pointer holding the CRL path.
- **type** type of certificate to be loaded.
- **monitor** requests monitoring in LoadCRL().

See:

- wolfSSL_CertManagerEnableCRL
- wolfSSL_LoadCRL
- wolfSSL_CertManagerFreeCRL

Return:

- SSL_SUCCESS if there is no error in wolfSSL_CertManagerLoadCRL and if LoadCRL returns successfully.
- BAD_FUNC_ARG if the WOLFSSL_CERT_MANAGER struct is NULL.
- SSL_FATAL_ERROR if wolfSSL_CertManagerEnableCRL returns anything other than SSL_SUCCESS.
- BAD_PATH_ERROR if the path is NULL.
- MEMORY_E if LoadCRL fails to allocate heap memory.

Example

```
#include <wolfssl/ssl.h>

int wolfSSL_LoadCRL(WOLFSSL* ssl, const char* path, int type,
int monitor);
…
wolfSSL_CertManagerLoadCRL(SSL_CM(ssl), path, type, monitor);
```

C.58.4.348 function wolfSSL_CertManagerLoadCRLBuffer

```
int wolfSSL_CertManagerLoadCRLBuffer(
    WOLFSSL_CERT_MANAGER * cm,
    const unsigned char * buff,
    long sz,
    int type
)
```

The function loads the CRL file by calling BufferLoadCRL.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure.
- **buff** a constant byte type and is the buffer.
- **sz** a long int representing the size of the buffer.
- **type** a long integer that holds the certificate type.

See:

- BufferLoadCRL
- wolfSSL_CertManagerEnableCRL

Return:

- SSL_SUCCESS returned if the function completed without errors.
- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER is NULL.
- SSL_FATAL_ERROR returned if there is an error associated with the WOLFSSL_CERT_MANAGER.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CERT_MANAGER* cm;
const unsigned char* buff;
long sz; size of buffer
int type;  cert type
...
int ret = wolfSSL_CertManagerLoadCRLBuffer(cm, buff, sz, type);
if(ret == SSL_SUCCESS){
return ret;
} else {
    Failure case.
}
```

C.58.4.349 function wolfSSL_CertManagerSetCRL_Cb

```
int wolfSSL_CertManagerSetCRL_Cb(
    WOLFSSL_CERT_MANAGER * cm,
    CbMissingCRL cb
)
```

This function sets the CRL Certificate Manager callback. If HAVE_CRL is defined and a matching CRL record is not found then the cbMissingCRL is called (set via wolfSSL_CertManagerSetCRL_Cb). This allows you to externally retrieve the CRL and load it.

Parameters:

- **cm** the WOLFSSL_CERT_MANAGER structure holding the information for the certificate.
- **cb** a function pointer to (*CbMissingCRL) that is set to the cbMissingCRL member of the WOLFSSL_CERT_MANAGER.

See:

- CbMissingCRL
- wolfSSL_SetCRL_Cb

Return:

- SSL_SUCCESS returned upon successful execution of the function and subroutines.
- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER structure is NULL.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = wolfSSL_CTX_new(protocol method);
WOLFSSL* ssl = wolfSSL_new(ctx);
…
void cb(const char* url){
    Function body.
}
…
CbMissingCRL cb = CbMissingCRL;
…
if(ctx){
    return wolfSSL_CertManagerSetCRL_Cb(SSL_CM(ssl), cb);
}
```

C.58.4.350 function wolfSSL_CertManagerSetCRLUpdate_Cb

```
int wolfSSL_CertManagerSetCRLUpdate_Cb(
    WOLFSSL_CERT_MANAGER * cm,
    CbUpdateCRL cb
)
```

This function sets the CRL Update callback. If HAVE_CRL and HAVE_CRL_UPDATE_CB is defined , and an entry with the same issuer and a lower CRL number exists when a CRL is added, then the CbUpdateCRL is called with the details of the existing entry and the new one replacing it.

Parameters:

- **cm** the WOLFSSL_CERT_MANAGER structure holding the information for the certificate.
- **cb** a function pointer to (*CbUpdateCRL) that is set to the cbUpdateCRL member of the WOLFSSL_CERT_MANAGER. Signature requirement: void (*CbUpdateCRL)(CrlInfo *old, CrlInfo *new);

See: CbUpdateCRL

Return:

- SSL_SUCCESS returned upon successful execution of the function and subroutines.
- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER structure is NULL.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = wolfSSL_CTX_new(protocol method);
WOLFSSL* ssl = wolfSSL_new(ctx);
…
void cb(CrlInfo *old, CrlInfo *new){
    Function body.
}
…
CbUpdateCRL cb = CbUpdateCRL;
…
if(ctx){
    return wolfSSL_CertManagerSetCRLUpdate_Cb(SSL_CM(ssl), cb);
}
```

C.58.4.351 function wolfSSL_CertManagerGetCRLInfo

```
int wolfSSL_CertManagerGetCRLInfo(
    WOLFSSL_CERT_MANAGER * cm,
    CrlInfo * info,
    const byte * buff,
    long sz,
    int type
)
```

This function yields a structure with parsed CRL information from an encoded CRL buffer.

Parameters:

- **cm** the WOLFSSL_CERT_MANAGER structure..
- **info** pointer to caller managed CrlInfo structure that will receive the CRL information.
- **buff** input buffer containing encoded CRL.
- **sz** the length in bytes of the input CRL data in buff.
- **type** WOLFSSL_FILETYPE_PEM or WOLFSSL_FILETYPE_DER
- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().

See:

- CbUpdateCRL
- wolfSSL_SetCRL_Cb
- wolfSSL_CertManagerLoadCRL

Return:

- SSL_SUCCESS returned upon successful execution of the function and subroutines.
- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER structure is NULL.
- SSL_SUCCESS returned upon successful execution of the function and subroutines.
- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER structure is NULL.

Example

```
#include <wolfssl/ssl.h>

CrlInfo info;
WOLFSSL_CERT_MANAGER* cm = NULL;

cm = wolfSSL_CertManagerNew();

// Read crl data from file into buffer

wolfSSL_CertManagerGetCRLInfo(cm, &info, crlData, crlDataLen,
                              WOLFSSL_FILETYPE_PEM);
```

This function frees the CRL stored in the Cert Manager. An application can update the CRL by calling wolfSSL_CertManagerFreeCRL and then loading the new CRL.

Example

```
#include <wolfssl/ssl.h>

const char* crl1     = "./certs/crl/crl.pem";
WOLFSSL_CERT_MANAGER* cm = NULL;

cm = wolfSSL_CertManagerNew();
wolfSSL_CertManagerLoadCRL(cm, crl1, WOLFSSL_FILETYPE_PEM, 0);
```

```
...
wolfSSL_CertManagerFreeCRL(cm);
```

C.58.4.352 function wolfSSL_CertManagerCheckOCSP

```
int wolfSSL_CertManagerCheckOCSP(
    WOLFSSL_CERT_MANAGER * cm,
    const unsigned char * der,
    int sz
)
```

The function enables the WOLFSSL_CERT_MANAGER's member, ocspEnabled to signify that the OCSP check option is enabled.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().
- **der** a byte pointer to the certificate.
- **sz** an int type representing the size of the DER cert.

See:

- ParseCertRelative
- CheckCertOCSP

Return:

- SSL_SUCCESS returned on successful execution of the function. The ocspEnabled member of the WOLFSSL_CERT_MANAGER is enabled.
- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER structure is NULL or if an argument value that is not allowed is passed to a subroutine.
- MEMORY_E returned if there is an error allocating memory within this function or a subroutine.

Example

```
#import <wolfssl/ssl.h>

WOLFSSL* ssl = wolfSSL_new(ctx);
byte* der;
int sz; size of der
...
if(wolfSSL_CertManagerCheckOCSP(cm, der, sz) != SSL_SUCCESS){
 Failure case.
}
```

C.58.4.353 function wolfSSL_CertManagerEnableOCSP

```
int wolfSSL_CertManagerEnableOCSP(
    WOLFSSL_CERT_MANAGER * cm,
    int options
)
```

Turns on OCSP if it's turned off and if compiled with the set option available.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().
- **options** used to set values in WOLFSSL_CERT_MANAGER struct.

See: wolfSSL_CertManagerNew

Return:

- SSL_SUCCESS returned if the function call is successful.
- BAD_FUNC_ARG if cm struct is NULL.
- MEMORY_E if WOLFSSL_OCSP struct value is NULL.
- SSL_FAILURE initialization of WOLFSSL_OCSP struct fails to initialize.
- NOT_COMPILED_IN build not compiled with correct feature enabled.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = wolfSSL_CTX_new(protocol method);
WOLFSSL* ssl = wolfSSL_new(ctx);
WOLFSSL_CERT_MANAGER* cm = wolfSSL_CertManagerNew();
int options;
…
if(wolfSSL_CertManagerEnableOCSP(SSL_CM(ssl), options) != SSL_SUCCESS){
    Failure case.
}
```

C.58.4.354 function wolfSSL_CertManagerDisableOCSP

```
int wolfSSL_CertManagerDisableOCSP(
    WOLFSSL_CERT_MANAGER * cm
)
```

Disables OCSP certificate revocation.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure.

See: wolfSSL_DisableCRL

Return:

- SSL_SUCCESS wolfSSL_CertMangerDisableCRL successfully disabled the crlEnabled member of the WOLFSSL_CERT_MANAGER structure.
- BAD_FUNC_ARG the WOLFSSL structure was NULL.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = wolfSSL_CTX_new(method);
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_CertManagerDisableOCSP(ssl) != SSL_SUCCESS){
    Fail case.
}
```

C.58.4.355 function wolfSSL_CertManagerSetOCSPOverrideURL

```
int wolfSSL_CertManagerSetOCSPOverrideURL(
    WOLFSSL_CERT_MANAGER * cm,
    const char * url
)
```

The function copies the url to the ocspOverrideURL member of the WOLFSSL_CERT_MANAGER structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- ocspOverrideURL
- wolfSSL_SetOCSP_OverrideURL

Return:

- SSL_SUCCESS the function was able to execute as expected.
- BAD_FUNC_ARG the WOLFSSL_CERT_MANAGER struct is NULL.
- MEMEORY_E Memory was not able to be allocated for the ocspOverrideURL member of the certificate manager.

Example

```
#include <wolfssl/ssl.h>
WOLFSSL_CERT_MANAGER* cm = wolfSSL_CertManagerNew();
const char* url;
…
int wolfSSL_SetOCSP_OverrideURL(WOLFSSL* ssl, const char* url)
…
if(wolfSSL_CertManagerSetOCSPOverrideURL(SSL_CM(ssl), url) != SSL_SUCCESS){
    Failure case.
}
```

C.58.4.356 function wolfSSL_CertManagerSetOCSP_Cb

```
int wolfSSL_CertManagerSetOCSP_Cb(
    WOLFSSL_CERT_MANAGER * cm,
    CbOCSPIO ioCb,
    CbOCSPRespFree respFreeCb,
    void * ioCbCtx
)
```

The function sets the OCSP callback in the WOLFSSL_CERT_MANAGER.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure.
- **ioCb** a function pointer of type CbOCSPIO.
- **respFreeCb** - a function pointer of type CbOCSPRespFree.
- **ioCbCtx** - a void pointer variable to the I/O callback user registered context.

See:

- wolfSSL_CertManagerSetOCSPOverrideURL
- wolfSSL_CertManagerCheckOCSP
- wolfSSL_CertManagerEnableOCSPStapling
- wolfSSL_EnableOCSP
- wolfSSL_DisableOCSP
- wolfSSL_SetOCSP_Cb

Return:

- SSL_SUCCESS returned on successful execution. The arguments are saved in the WOLFSSL_CERT_MANAGER structure.

- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER is NULL.

Example

```
#include <wolfssl/ssl.h>

wolfSSL_SetOCSP_Cb(WOLFSSL* ssl, CbOCSPIO ioCb,
CbOCSPRespFree respFreeCb, void* ioCbCtx){
…
return wolfSSL_CertManagerSetOCSP_Cb(SSL_CM(ssl), ioCb, respFreeCb, ioCbCtx);
```

C.58.4.357 function wolfSSL_CertManagerEnableOCSPStapling

```
int wolfSSL_CertManagerEnableOCSPStapling(
    WOLFSSL_CERT_MANAGER * cm
)
```

This function turns on OCSP stapling if it is not turned on as well as set the options.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, a member of the WOLFSSL_CTX structure.

See: wolfSSL_CTX_EnableOCSPStapling

Return:

- SSL_SUCCESS returned if there were no errors and the function executed successfully.
- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER structure is NULL or otherwise if there was a unpermitted argument value passed to a subroutine.
- MEMORY_E returned if there was an issue allocating memory.
- SSL_FAILURE returned if the initialization of the OCSP structure failed.
- NOT_COMPILED_IN returned if wolfSSL was not compiled with HAVE_CERTIFICATE_STATUS_REQUEST option.

Example

```
int wolfSSL_CTX_EnableOCSPStapling(WOLFSSL_CTX* ctx){
…
return wolfSSL_CertManagerEnableOCSPStapling(ctx->cm);
```

C.58.4.358 function wolfSSL_EnableCRL

```
int wolfSSL_EnableCRL(
    WOLFSSL * ssl,
    int options
)
```

Enables CRL certificate revocation.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **options** an integer that is used to determine the setting of crlCheckAll member of the WOLFSSL_CERT_MANAGER structure.

See:

- wolfSSL_CertManagerEnableCRL
- InitCRL

Return:

- SSL_SUCCESS the function and subroutines returned with no errors.
- BAD_FUNC_ARG returned if the WOLFSSL structure is NULL.
- MEMORY_E returned if the allocation of memory failed.
- SSL_FAILURE returned if the InitCRL function does not return successfully.
- NOT_COMPILED_IN HAVE_CRL was not enabled during the compiling.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
…
if (wolfSSL_EnableCRL(ssl, WOLFSSL_CRL_CHECKALL) != SSL_SUCCESS){
    // Failure case. SSL_SUCCESS was not returned by this function or
a subroutine
}
```

C.58.4.359 function wolfSSL_DisableCRL

```
int wolfSSL_DisableCRL(
    WOLFSSL * ssl
)
```

Disables CRL certificate revocation.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_CertManagerDisableCRL
- wolfSSL_CertManagerDisableOCSP

Return:

- SSL_SUCCESS wolfSSL_CertMangerDisableCRL successfully disabled the crlEnabled member of the WOLFSSL_CERT_MANAGER structure.
- BAD_FUNC_ARG the WOLFSSL structure was NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_DisableCRL(ssl) != SSL_SUCCESS){
    // Failure case
}
```

C.58.4.360 function wolfSSL_LoadCRL

```
int wolfSSL_LoadCRL(
    WOLFSSL * ssl,
    const char * path,
    int type,
    int monitor
)
```

A wrapper function that ends up calling LoadCRL to load the certificate for revocation checking.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

- **path** a constant character pointer that holds the path to the crl file.
- **type** an integer representing the type of certificate.
- **monitor** an integer variable used to verify the monitor path if requested.

See:

- wolfSSL_CertManagerLoadCRL
- wolfSSL_CertManagerEnableCRL
- LoadCRL

Return:

- WOLFSSL_SUCCESS returned if the function and all of the subroutines executed without error.
- SSL_FATAL_ERROR returned if one of the subroutines does not return successfully.
- BAD_FUNC_ARG if the WOLFSSL_CERT_MANAGER or the WOLFSSL structure are NULL.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
const char* crlPemDir;
…
if(wolfSSL_LoadCRL(ssl, crlPemDir, SSL_FILETYPE_PEM, 0) != SSL_SUCCESS){
    // Failure case. Did not return SSL_SUCCESS.
}
```

C.58.4.361 function wolfSSL_SetCRL_Cb

```
int wolfSSL_SetCRL_Cb(
    WOLFSSL * ssl,
    CbMissingCRL cb
)
```

Sets the CRL callback in the WOLFSSL_CERT_MANAGER structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **cb** a function pointer to CbMissingCRL.

See:

- CbMissingCRL
- wolfSSL_CertManagerSetCRL_Cb

Return:

- SSL_SUCCESS returned if the function or subroutine executes without error. The cbMissingCRL member of the WOLFSSL_CERT_MANAGER is set.
- BAD_FUNC_ARG returned if the WOLFSSL or WOLFSSL_CERT_MANAGER structure is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
…
void cb(const char* url) // required signature
{
    // Function body
}
…
int crlCb = wolfSSL_SetCRL_Cb(ssl, cb);
if(crlCb != SSL_SUCCESS){
```

```
    // The callback was not set properly
}
```

C.58.4.362 function wolfSSL_EnableOCSP

```
int wolfSSL_EnableOCSP(
    WOLFSSL * ssl,
    int options
)
```

This function enables OCSP certificate verification. The value of options if formed by or'ing one or more of the following options: WOLFSSL_OCSP_URL_OVERRIDE - use the override URL instead of the URL in certificates. The override URL is specified using the wolfSSL_CTX_SetOCSP_OverrideURL() function. WOLFSSL_OCSP_CHECKALL - Set all OCSP checks on WOLFSSL_OCSP_NO_NONCE - Set nonce option for creating OCSP requests.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **options** an integer type passed to wolfSSL_CertMangerENableOCSP() used for settings check.

See: wolfSSL_CertManagerEnableOCSP

Return:

- SSL_SUCCESS returned if the function and subroutines executes without errors.
- BAD_FUNC_ARG returned if an argument in this function or any subroutine receives an invalid argument value.
- MEMORY_E returned if there was an error allocating memory for a structure or other variable.
- NOT_COMPILED_IN returned if wolfSSL was not compiled with the HAVE_OCSP option.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
int options; // initialize to option constant
…
int ret = wolfSSL_EnableOCSP(ssl, options);
if(ret != SSL_SUCCESS){
    // OCSP is not enabled
}
```

C.58.4.363 function wolfSSL_DisableOCSP

```
int wolfSSL_DisableOCSP(
    WOLFSSL * ssl
)
```

Disables the OCSP certificate revocation option.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_CertManagerDisableOCSP

Return:

- SSL_SUCCESS returned if the function and its subroutine return with no errors. The ocspEnabled member of the WOLFSSL_CERT_MANAGER structure was successfully set.
- BAD_FUNC_ARG returned if the WOLFSSL structure is NULL.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
…
if(wolfSSL_DisableOCSP(ssl) != SSL_SUCCESS){
    // Returned with an error. Failure case in this block.
}
```

C.58.4.364 function wolfSSL_SetOCSP_OverrideURL

```
int wolfSSL_SetOCSP_OverrideURL(
    WOLFSSL * ssl,
    const char * url
)
```

This function sets the ocspOverrideURL member in the WOLFSSL_CERT_MANAGER structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **url** a constant char pointer to the url that will be stored in the ocspOverrideURL member of the WOLFSSL_CERT_MANAGER structure.

See: wolfSSL_CertManagerSetOCSPOverrideURL

Return:

- SSL_SUCCESS returned on successful execution of the function.
- BAD_FUNC_ARG returned if the WOLFSSL struct is NULL or if a unpermitted argument was passed to a subroutine.
- MEMORY_E returned if there was an error allocating memory in the subroutine.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
char url[URLSZ];
...
if(wolfSSL_SetOCSP_OverrideURL(ssl, url)){
    // The override url is set to the new value
}
```

C.58.4.365 function wolfSSL_SetOCSP_Cb

```
int wolfSSL_SetOCSP_Cb(
    WOLFSSL * ssl,
    CbOCSPIO ioCb,
    CbOCSPRespFree respFreeCb,
    void * ioCbCtx
)
```

This function sets the OCSP callback in the WOLFSSL_CERT_MANAGER structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **ioCb** a function pointer to type CbOCSPIO.
- **respFreeCb** a function pointer to type CbOCSPRespFree which is the call to free the response memory.
- **ioCbCtx** a void pointer that will be held in the ocspIOCtx member of the CM.

See:

- wolfSSL_CertManagerSetOCSP_Cb
- CbOCSPIO
- CbOCSPRespFree

Return:

- SSL_SUCCESS returned if the function executes without error. The ocspIOCb, ocspRespFreeCb, and ocspIOCtx members of the CM are set.
- BAD_FUNC_ARG returned if the WOLFSSL or WOLFSSL_CERT_MANAGER structures are NULL.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
…
int OCSPIO_CB(void* , const char*, int , unsigned char* , int,
unsigned char**){  // must have this signature
// Function Body
}
…
void OCSPRespFree_CB(void* , unsigned char* ){ // must have this signature
    // function body
}
…
void* ioCbCtx;
CbOCSPRespFree CB_OCSPRespFree;

if(wolfSSL_SetOCSP_Cb(ssl, OCSPIO_CB( pass args ), CB_OCSPRespFree,
            ioCbCtx) != SSL_SUCCESS){
    // Callback not set
}
```

C.58.4.366 function wolfSSL_CTX_EnableCRL

```
int wolfSSL_CTX_EnableCRL(
    WOLFSSL_CTX * ctx,
    int options
)
```

Enables CRL certificate verification through the CTX.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **options** option flags for enabling CRL.

See:

- wolfSSL_CertManagerEnableCRL
- InitCRL
- wolfSSL_CTX_DisableCRL

Return:

- SSL_SUCCESS returned if this function and it's subroutines execute without errors.
- BAD_FUNC_ARG returned if the CTX struct is NULL or there was otherwise an invalid argument passed in a subroutine.
- MEMORY_E returned if there was an error allocating memory during execution of the function.

- SSL_FAILURE returned if the crl member of the WOLFSSL_CERT_MANAGER fails to initialize correctly.
- BAD_MUTEX_E returned if locking the certificate manager failed.
- NOT_COMPILED_IN wolfSSL was not compiled with the HAVE_CRL option.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_CTX_EnableCRL(ssl->ctx, options) != SSL_SUCCESS){
    // The function failed
}
```

C.58.4.367 function wolfSSL_CTX_DisableCRL

```
int wolfSSL_CTX_DisableCRL(
    WOLFSSL_CTX * ctx
)
```

This function disables CRL verification in the CTX structure.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

See: wolfSSL_CertManagerDisableCRL

Return:

- SSL_SUCCESS returned if the function executes without error. The crlEnabled member of the WOLFSSL_CERT_MANAGER struct is set to 0.
- BAD_FUNC_ARG returned if either the CTX struct or the CM struct has a NULL value.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_CTX_DisableCRL(ssl->ctx) != SSL_SUCCESS){
    // Failure case.
}
```

C.58.4.368 function wolfSSL_CTX_LoadCRL

```
int wolfSSL_CTX_LoadCRL(
    WOLFSSL_CTX * ctx,
    const char * path,
    int type,
    int monitor
)
```

This function loads CRL into the WOLFSSL_CTX structure through wolfSSL_CertManagerLoadCRL().

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **path** the path to the certificate.
- **type** an integer variable holding the type of certificate.
- **monitor** an integer variable used to determine if the monitor path is requested.

See:

- wolfSSL_CertManagerLoadCRL
- LoadCRL

Return:

- SSL_SUCCESS - returned if the function and its subroutines execute without error.
- BAD_FUNC_ARG - returned if this function or any subroutines are passed NULL structures.
- BAD_PATH_ERROR - returned if the path variable opens as NULL.
- MEMORY_E - returned if an allocation of memory failed.

Example

```
WOLFSSL_CTX* ctx;
const char* path;
…
return wolfSSL_CTX_LoadCRL(ctx, path, SSL_FILETYPE_PEM, 0);
```

C.58.4.369 function wolfSSL_CTX_SetCRL_Cb

```
int wolfSSL_CTX_SetCRL_Cb(
    WOLFSSL_CTX * ctx,
    CbMissingCRL cb
)
```

This function will set the callback argument to the cbMissingCRL member of the WOLFSSL_CERT_MANAGER structure by calling wolfSSL_CertManagerSetCRL_Cb.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().
- **cb** a pointer to a callback function of type CbMissingCRL. Signature requirement: void (*CbMissingCRL)(const char* url);

See:

- wolfSSL_CertManagerSetCRL_Cb
- CbMissingCRL

Return:

- SSL_SUCCESS returned for a successful execution. The WOLFSSL_CERT_MANAGER structure's member cbMssingCRL was successfully set to cb.
- BAD_FUNC_ARG returned if WOLFSSL_CTX or WOLFSSL_CERT_MANAGER are NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
…
void cb(const char* url) // Required signature
{
    // Function body
}
…
if (wolfSSL_CTX_SetCRL_Cb(ctx, cb) != SSL_SUCCESS){
    // Failure case, cb was not set correctly.
}
```

C.58.4.370 function wolfSSL_CTX_EnableOCSP

```
int wolfSSL_CTX_EnableOCSP(
    WOLFSSL_CTX * ctx,
    int options
)
```

This function sets options to configure behavior of OCSP functionality in wolfSSL. The value of options if formed by or'ing one or more of the following options: WOLFSSL_OCSP_URL_OVERRIDE - use the override URL instead of the URL in certificates. The override URL is specified using the wolfSSL_CTX_SetOCSP_OverrideURL() function. WOLFSSL_OCSP_CHECKALL - Set all OCSP checks on WOLFSSL_OCSP_NO_NONCE - Set nonce option for creating OCSP requests.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **options** value used to set the OCSP options.

See:

- wolfSSL_CertManagerEnableOCSP
- wolfSSL_EnableOCSP

Return:

- SSL_SUCCESS is returned upon success.
- SSL_FAILURE is returned upon failure.
- BAD_MUTEX_E returned if locking the certificate manager failed.
- NOT_COMPILED_IN is returned when this function has been called, but OCSP support was not enabled when wolfSSL was compiled.
- BAD_FUNC_ARG will be returned when ctx is NULL.

This function only sets the OCSP options when wolfSSL has been compiled with OCSP support (–enable-ocsp, #define HAVE_OCSP).

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
int options; // initialize to option constant
…
int ret = wolfSSL_CTX_EnableOCSP(ctx, options);
if(ret != SSL_SUCCESS){
    // OCSP is not enabled
}
```

C.58.4.371 function wolfSSL_CTX_DisableOCSP

```
int wolfSSL_CTX_DisableOCSP(
    WOLFSSL_CTX * ctx
)
```

This function disables OCSP certificate revocation checking by affecting the ocspEnabled member of the WOLFSSL_CERT_MANAGER structure.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

See:

- wolfSSL_DisableOCSP
- wolfSSL_CertManagerDisableOCSP

Return:

- SSL_SUCCESS returned if the function executes without error. The ocspEnabled member of the CM has been disabled.
- BAD_FUNC_ARG returned if the WOLFSSL_CTX structure is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(!wolfSSL_CTX_DisableOCSP(ssl->ctx)){
    // OCSP is not disabled
}
```

C.58.4.372 function wolfSSL_CTX_SetOCSP_OverrideURL

```
int wolfSSL_CTX_SetOCSP_OverrideURL(
    WOLFSSL_CTX * ctx,
    const char * url
)
```

This function manually sets the URL for OCSP to use. By default, OCSP will use the URL found in the individual certificate unless the WOLFSSL_OCSP_URL_OVERRIDE option is set using the wolfSSL_CTX_EnableOCSP.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **url** pointer to the OCSP URL for wolfSSL to use.

See: wolfSSL_CTX_OCSP_set_options

Return:

- SSL_SUCCESS is returned upon success.
- SSL_FAILURE is returned upon failure.
- NOT_COMPILED_IN is returned when this function has been called, but OCSP support was not enabled when wolfSSL was compiled.
- BAD_FUNC_ARG will be returned when ctx is NULL.

Example

```
WOLFSSL_CTX* ctx = 0;
...
wolfSSL_CTX_OCSP_set_override_url(ctx, “custom-url-here”);
```

C.58.4.373 function wolfSSL_CTX_SetOCSP_Cb

```
int wolfSSL_CTX_SetOCSP_Cb(
    WOLFSSL_CTX * ctx,
    CbOCSPIO ioCb,
    CbOCSPRespFree respFreeCb,
    void * ioCbCtx
)
```

Sets the callback for the OCSP in the WOLFSSL_CTX structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

- **ioCb** a CbOCSPIO type that is a function pointer.
- **respFreeCb** a CbOCSPRespFree type that is a function pointer.
- **ioCbCtx** a void pointer that will be held in the WOLFSSL_CERT_MANAGER.

See:

- wolfSSL_CertManagerSetOCSP_Cb
- CbOCSPIO
- CbOCSPRespFree

Return:

- SSL_SUCCESS returned if the function executed successfully. The ocspIOCb, ocspRespFreeCb, and ocspIOCtx members in the CM were successfully set.
- BAD_FUNC_ARG returned if the WOLFSSL_CTX or WOLFSSL_CERT_MANAGER structure is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
…
CbOCSPIO ocspIOCb;
CbOCSPRespFree ocspRespFreeCb;
…
void* ioCbCtx;

int isSetOCSP = wolfSSL_CTX_SetOCSP_Cb(ctx, ocspIOCb,
ocspRespFreeCb, ioCbCtx);

if(isSetOCSP != SSL_SUCCESS){
    // The function did not return successfully.
}
```

C.58.4.374 function wolfSSL_CTX_EnableOCSPStapling

```
int wolfSSL_CTX_EnableOCSPStapling(
    WOLFSSL_CTX * ctx
)
```

This function enables OCSP stapling by calling wolfSSL_CertManagerEnableOCSPStapling().

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

See:

- wolfSSL_CertManagerEnableOCSPStapling
- InitOCSP

Return:

- SSL_SUCCESS returned if there were no errors and the function executed successfully.
- BAD_FUNC_ARG returned if the WOLFSSL_CTX structure is NULL or otherwise if there was a unpermitted argument value passed to a subroutine.
- MEMORY_E returned if there was an issue allocating memory.
- SSL_FAILURE returned if the initialization of the OCSP structure failed.
- BAD_MUTEX_E returned if locking the certificate manager failed.
- NOT_COMPILED_IN returned if wolfSSL was not compiled with HAVE_CERTIFICATE_STATUS_REQUEST option.

Example

```
WOLFSSL* ssl = WOLFSSL_new();
ssl->method.version; // set to desired protocol
...
if(!wolfSSL_CTX_EnableOCSPStapling(ssl->ctx)){
    // OCSP stapling is not enabled
}
```

C.58.4.375 function wolfSSL_KeepArrays

```
void wolfSSL_KeepArrays(
    WOLFSSL * ssl
)
```

Normally, at the end of the SSL handshake, wolfSSL frees temporary arrays. Calling this function before the handshake begins will prevent wolfSSL from freeing temporary arrays. Temporary arrays may be needed for things such as wolfSSL_get_keys() or PSK hints. When the user is done with temporary arrays, either wolfSSL_FreeArrays() may be called to free the resources immediately, or alternatively the resources will be freed when the associated SSL object is freed.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_FreeArrays

Return: none No return.

Example

```
WOLFSSL* ssl;
...
wolfSSL_KeepArrays(ssl);
```

C.58.4.376 function wolfSSL_FreeArrays

```
void wolfSSL_FreeArrays(
    WOLFSSL * ssl
)
```

Normally, at the end of the SSL handshake, wolfSSL frees temporary arrays. If wolfSSL_KeepArrays() has been called before the handshake, wolfSSL will not free temporary arrays. This function explicitly frees temporary arrays and should be called when the user is done with temporary arrays and does not want to wait for the SSL object to be freed to free these resources.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_KeepArrays

Return: none No return.

Example

```
WOLFSSL* ssl;
...
wolfSSL_FreeArrays(ssl);
```

C.58.4.377 function wolfSSL_UseSNI

```
int wolfSSL_UseSNI(
    WOLFSSL * ssl,
    unsigned char type,
    const void * data,
    unsigned short size
)
```

This function enables the use of Server Name Indication in the SSL object passed in the 'ssl' parameter. It means that the SNI extension will be sent on ClientHello by wolfSSL client and wolfSSL server will respond ClientHello + SNI with either ServerHello + blank SNI or alert fatal in case of SNI mismatch.

Parameters:

- **ssl** pointer to a SSL object, created with wolfSSL_new().
- **type** indicates which type of server name is been passed in data. The known types are: enum { WOLFSSL_SNI_HOST_NAME = 0 };
- **data** pointer to the server name data.
- **size** size of the server name data.

See:

- wolfSSL_new
- wolfSSL_CTX_UseSNI

Return:

- WOLFSSL_SUCCESS upon success.
- BAD_FUNC_ARG is the error that will be returned in one of these cases: ssl is NULL, data is NULL, type is a unknown value. (see below)
- MEMORY_E is the error returned when there is not enough memory.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;
WOLFSSL* ssl = 0;
ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // context creation failed
}
ssl = wolfSSL_new(ctx);
if (ssl == NULL) {
    // ssl creation failed
}
ret = wolfSSL_UseSNI(ssl, WOLFSSL_SNI_HOST_NAME, "www.yassl.com",
    strlen("www.yassl.com"));
if (ret != WOLFSSL_SUCCESS) {
    // sni usage failed
}
```

C.58.4.378 function wolfSSL_CTX_UseSNI

```
int wolfSSL_CTX_UseSNI(
    WOLFSSL_CTX * ctx,
    unsigned char type,
    const void * data,
```

```
    unsigned short size
)
```

This function enables the use of Server Name Indication for SSL objects created from the SSL context passed in the 'ctx' parameter. It means that the SNI extension will be sent on ClientHello by wolfSSL clients and wolfSSL servers will respond ClientHello + SNI with either ServerHello + blank SNI or alert fatal in case of SNI mismatch.

Parameters:

- **ctx** pointer to a SSL context, created with wolfSSL_CTX_new().
- **type** indicates which type of server name is been passed in data. The known types are: enum { WOLFSSL_SNI_HOST_NAME = 0 };
- **data** pointer to the server name data.
- **size** size of the server name data.

See:

- wolfSSL_CTX_new
- wolfSSL_UseSNI

Return:

- WOLFSSL_SUCCESS upon success.
- BAD_FUNC_ARG is the error that will be returned in one of these cases: ctx is NULL, data is NULL, type is a unknown value. (see below)
- MEMORY_E is the error returned when there is not enough memory.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;
ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // context creation failed
}
ret = wolfSSL_CTX_UseSNI(ctx, WOLFSSL_SNI_HOST_NAME, "www.yassl.com",
    strlen("www.yassl.com"));
if (ret != WOLFSSL_SUCCESS) {
    // sni usage failed
}
```

C.58.4.379 function wolfSSL_SNI_SetOptions

```
void wolfSSL_SNI_SetOptions(
    WOLFSSL * ssl,
    unsigned char type,
    unsigned char options
)
```

This function is called on the server side to configure the behavior of the SSL session using Server Name Indication in the SSL object passed in the 'ssl' parameter. The options are explained below.

Parameters:

- **ssl** pointer to a SSL object, created with wolfSSL_new().
- **type** indicates which type of server name is been passed in data. The known types are: enum { WOLFSSL_SNI_HOST_NAME = 0 };
- **options** a bitwise semaphore with the chosen options. The available options are: enum { WOLFSSL_SNI_CONTINUE_ON_MISMATCH = 0x01, WOLFSSL_SNI_ANSWER_ON_MISMATCH = 0x02 };

Normally the server will abort the handshake by sending a fatal-level unrecognized_name(112) alert if the hostname provided by the client mismatch with the servers.
- **WOLFSSL_SNI_CONTINUE_ON_MISMATCH** With this option set, the server will not send a SNI response instead of aborting the session.
- **WOLFSSL_SNI_ANSWER_ON_MISMATCH** - With this option set, the server will send a SNI response as if the host names match instead of aborting the session.

See:

- wolfSSL_new
- wolfSSL_UseSNI
- wolfSSL_CTX_SNI_SetOptions

Return: none No returns.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;
WOLFSSL* ssl = 0;
ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // context creation failed
}
ssl = wolfSSL_new(ctx);
if (ssl == NULL) {
    // ssl creation failed
}
ret = wolfSSL_UseSNI(ssl, 0, "www.yassl.com", strlen("www.yassl.com"));
if (ret != WOLFSSL_SUCCESS) {
    // sni usage failed
}
wolfSSL_SNI_SetOptions(ssl, WOLFSSL_SNI_HOST_NAME,
    WOLFSSL_SNI_CONTINUE_ON_MISMATCH);
```

C.58.4.380 function wolfSSL_CTX_SNI_SetOptions

```
void wolfSSL_CTX_SNI_SetOptions(
    WOLFSSL_CTX * ctx,
    unsigned char type,
    unsigned char options
)
```

This function is called on the server side to configure the behavior of the SSL sessions using Server Name Indication for SSL objects created from the SSL context passed in the 'ctx' parameter. The options are explained below.

Parameters:

- **ctx** pointer to a SSL context, created with wolfSSL_CTX_new().
- **type** indicates which type of server name is been passed in data. The known types are: enum { WOLFSSL_SNI_HOST_NAME = 0 };
- **options** a bitwise semaphore with the chosen options. The available options are: enum { WOLFSSL_SNI_CONTINUE_ON_MISMATCH = 0x01, WOLFSSL_SNI_ANSWER_ON_MISMATCH = 0x02 }; Normally the server will abort the handshake by sending a fatal-level unrecognized_name(112) alert if the hostname provided by the client mismatch with the servers.
- **WOLFSSL_SNI_CONTINUE_ON_MISMATCH** With this option set, the server will not send a SNI response instead of aborting the session.

- **WOLFSSL_SNI_ANSWER_ON_MISMATCH** With this option set, the server will send a SNI response as if the host names match instead of aborting the session.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_UseSNI
- wolfSSL_SNI_SetOptions

Return: none No returns.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;
ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
   // context creation failed
}
ret = wolfSSL_CTX_UseSNI(ctx, 0, "www.yassl.com", strlen("www.yassl.com"));
if (ret != WOLFSSL_SUCCESS) {
    // sni usage failed
}
wolfSSL_CTX_SNI_SetOptions(ctx, WOLFSSL_SNI_HOST_NAME,
WOLFSSL_SNI_CONTINUE_ON_MISMATCH);
```

C.58.4.381 function wolfSSL_SNI_GetFromBuffer

```
int wolfSSL_SNI_GetFromBuffer(
    const unsigned char * clientHello,
    unsigned int helloSz,
    unsigned char type,
    unsigned char * sni,
    unsigned int * inOutSz
)
```

This function is called on the server side to retrieve the Server Name Indication provided by the client from the Client Hello message sent by the client to start a session. It does not requires context or session setup to retrieve the SNI.

Parameters:

- **buffer** pointer to the data provided by the client (Client Hello).
- **bufferSz** size of the Client Hello message.
- **type** indicates which type of server name is been retrieved from the buffer. The known types are: enum { WOLFSSL_SNI_HOST_NAME = 0 };
- **sni** pointer to where the output is going to be stored.
- **inOutSz** pointer to the output size, this value will be updated to MIN("SNI's length", inOutSz).

See:

- wolfSSL_UseSNI
- wolfSSL_CTX_UseSNI
- wolfSSL_SNI_GetRequest

Return:

- WOLFSSL_SUCCESS upon success.
- BAD_FUNC_ARG is the error that will be returned in one of this cases: buffer is NULL, bufferSz <= 0, sni is NULL, inOutSz is NULL or <= 0

- BUFFER_ERROR is the error returned when there is a malformed Client Hello message.
- INCOMPLETE_DATA is the error returned when there is not enough data to complete the extraction.

Example

```
unsigned char buffer[1024] = {0};
unsigned char result[32]   = {0};
int           length       = 32;
// read Client Hello to buffer...
ret = wolfSSL_SNI_GetFromBuffer(buffer, sizeof(buffer), 0, result, &length));
if (ret != WOLFSSL_SUCCESS) {
    // sni retrieve failed
}
```

C.58.4.382 function wolfSSL_SNI_Status

```
unsigned char wolfSSL_SNI_Status(
    WOLFSSL * ssl,
    unsigned char type
)
```

This function gets the status of an SNI object.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **type** the SNI type.

See:

- TLSX_SNI_Status
- TLSX_SNI_find
- TLSX_Find

Return:

- value This function returns the byte value of the SNI struct's status member if the SNI is not NULL.
- 0 if the SNI object is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
…
#define AssertIntEQ(x, y) AssertInt(x, y, ==, !=)
…
Byte type = WOLFSSL_SNI_HOST_NAME;
char* request = (char*)&type;
AssertIntEQ(WOLFSSL_SNI_NO_MATCH, wolfSSL_SNI_Status(ssl, type));
…
```

C.58.4.383 function wolfSSL_SNI_GetRequest

```
unsigned short wolfSSL_SNI_GetRequest(
    WOLFSSL * ssl,
    unsigned char type,
    void ** data
)
```

This function is called on the server side to retrieve the Server Name Indication provided by the client in a SSL session.

Parameters:

- **ssl** pointer to a SSL object, created with wolfSSL_new().
- **type** indicates which type of server name is been retrieved in data. The known types are: enum { WOLFSSL_SNI_HOST_NAME = 0 };
- **data** pointer to the data provided by the client.

See:

- wolfSSL_UseSNI
- wolfSSL_CTX_UseSNI

Return: size the size of the provided SNI data.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;
WOLFSSL* ssl = 0;
ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // context creation failed
}
ssl = wolfSSL_new(ctx);
if (ssl == NULL) {
    // ssl creation failed
}
ret = wolfSSL_UseSNI(ssl, 0, "www.yassl.com", strlen("www.yassl.com"));
if (ret != WOLFSSL_SUCCESS) {
    // sni usage failed
}
if (wolfSSL_accept(ssl) == SSL_SUCCESS) {
    void *data = NULL;
    unsigned short size = wolfSSL_SNI_GetRequest(ssl, 0, &data);
}
```

C.58.4.384 function wolfSSL_UseALPN

```
int wolfSSL_UseALPN(
    WOLFSSL * ssl,
    char * protocol_name_list,
    unsigned int protocol_name_listSz,
    unsigned char options
)
```

Setup ALPN use for a wolfSSL session.

Parameters:

- **ssl** The wolfSSL session to use.
- **protocol_name_list** List of protocol names to use. Comma delimited string is required.
- **protocol_name_listSz** Size of the list of protocol names.
- **options** WOLFSSL_ALPN_CONTINUE_ON_MISMATCH or WOLFSSL_ALPN_FAILED_ON_MISMATCH.

See: TLSX_UseALPN

Return:

- WOLFSSL_SUCCESS: upon success.
- BAD_FUNC_ARG Returned if ssl or protocol_name_list is null or protocol_name_listSz is too large or options contain something not supported.
- MEMORY_ERROR Error allocating memory for protocol list.
- SSL_FAILURE upon failure.

Example

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
WOLFSSL_METHOD method = // Some wolfSSL method
ctx = wolfSSL_CTX_new(method);
ssl = wolfSSL_new(ctx);

char alpn_list[] = {};

if (wolfSSL_UseALPN(ssl, alpn_list, sizeof(alpn_list),
    WOLFSSL_APN_FAILED_ON_MISMATCH) != WOLFSSL_SUCCESS)
{
   // Error setting session ticket
}
```

C.58.4.385 function wolfSSL_ALPN_GetProtocol

```
int wolfSSL_ALPN_GetProtocol(
    WOLFSSL * ssl,
    char ** protocol_name,
    unsigned short * size
)
```

This function gets the protocol name set by the server.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **protocol_name** a pointer to a char that represents the protocol name and will be held in the ALPN structure.
- **size** a word16 type that represents the size of the protocol_name.

See:

- TLSX_ALPN_GetRequest
- TLSX_Find

Return:

- SSL_SUCCESS returned on successful execution where no errors were thrown.
- SSL_FATAL_ERROR returned if the extension was not found or if there was no protocol match with peer. There will also be an error thrown if there is more than one protocol name accepted.
- SSL_ALPN_NOT_FOUND returned signifying that no protocol match with peer was found.
- BAD_FUNC_ARG returned if there was a NULL argument passed into the function.

Example

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new( protocol method );
WOLFSSL* ssl = WOLFSSL_new(ctx);
...
int err;
char* protocol_name = NULL;
```

```
Word16 protocol_nameSz = 0;
err = wolfSSL_ALPN_GetProtocol(ssl, &protocol_name, &protocol_nameSz);

if(err == SSL_SUCCESS){
    // Sent ALPN protocol
}
```

C.58.4.386 function wolfSSL_ALPN_GetPeerProtocol

```
int wolfSSL_ALPN_GetPeerProtocol(
    WOLFSSL * ssl,
    char ** list,
    unsigned short * listSz
)
```

This function copies the alpn_client_list data from the SSL object to the buffer.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **list** a pointer to the buffer. The data from the SSL object will be copied into it.
- **listSz** the buffer size.

See: wolfSSL_UseALPN

Return:

- SSL_SUCCESS returned if the function executed without error. The alpn_client_list member of the SSL object has been copied to the list parameter.
- BAD_FUNC_ARG returned if the list or listSz parameter is NULL.
- BUFFER_ERROR returned if there will be a problem with the list buffer (either it's NULL or the size is 0).
- MEMORY_ERROR returned if there was a problem dynamically allocating memory.

Example

```
#import <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method);
WOLFSSL* ssl = wolfSSL_new(ctx);
…
#ifdef HAVE_ALPN
char* list = NULL;
word16 listSz = 0;
…
err = wolfSSL_ALPN_GetPeerProtocol(ssl, &list, &listSz);

if(err == SSL_SUCCESS){
    List of protocols names sent by client
}
```

C.58.4.387 function wolfSSL_UseMaxFragment

```
int wolfSSL_UseMaxFragment(
    WOLFSSL * ssl,
    unsigned char mfl
)
```

This function is called on the client side to enable the use of Maximum Fragment Length in the SSL object passed in the 'ssl' parameter. It means that the Maximum Fragment Length extension will be sent on ClientHello by wolfSSL clients.

Parameters:

- **ssl** pointer to a SSL object, created with wolfSSL_new().
- **mfl** indicates which is the Maximum Fragment Length requested for the session. The available options are: enum { WOLFSSL_MFL_2_9 = 1, 512 bytes WOLFSSL_MFL_2_10 = 2, 1024 bytes WOLFSSL_MFL_2_11 = 3, 2048 bytes WOLFSSL_MFL_2_12 = 4, 4096 bytes WOLFSSL_MFL_2_13 = 5, 8192 bytes wolfSSL ONLY!!! };

See:

- wolfSSL_new
- wolfSSL_CTX_UseMaxFragment

Return:

- SSL_SUCCESS upon success.
- BAD_FUNC_ARG is the error that will be returned in one of these cases: ssl is NULL, mfl is out of range.
- MEMORY_E is the error returned when there is not enough memory.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;
WOLFSSL* ssl = 0;
ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // context creation failed
}
ssl = wolfSSL_new(ctx);
if (ssl == NULL) {
    // ssl creation failed
}
ret = wolfSSL_UseMaxFragment(ssl, WOLFSSL_MFL_2_11);
if (ret != 0) {
    // max fragment usage failed
}
```

C.58.4.388 function wolfSSL_CTX_UseMaxFragment

```
int wolfSSL_CTX_UseMaxFragment(
    WOLFSSL_CTX * ctx,
    unsigned char mfl
)
```

This function is called on the client side to enable the use of Maximum Fragment Length for SSL objects created from the SSL context passed in the 'ctx' parameter. It means that the Maximum Fragment Length extension will be sent on ClientHello by wolfSSL clients.

Parameters:

- **ctx** pointer to a SSL context, created with wolfSSL_CTX_new().
- **mfl** indicates which is the Maximum Fragment Length requested for the session. The available options are: enum { WOLFSSL_MFL_2_9 = 1 512 bytes, WOLFSSL_MFL_2_10 = 2 1024 bytes, WOLFSSL_MFL_2_11 = 3 2048 bytes WOLFSSL_MFL_2_12 = 4 4096 bytes, WOLFSSL_MFL_2_13 = 5 8192 bytes wolfSSL ONLY!!!, WOLFSSL_MFL_2_13 = 6 256 bytes wolfSSL ONLY!!! };

See:

- wolfSSL_CTX_new
- wolfSSL_UseMaxFragment

Return:

- SSL_SUCCESS upon success.
- BAD_FUNC_ARG is the error that will be returned in one of these cases: ctx is NULL, mfl is out of range.
- MEMORY_E is the error returned when there is not enough memory.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;
ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // context creation failed
}
ret = wolfSSL_CTX_UseMaxFragment(ctx, WOLFSSL_MFL_2_11);
if (ret != 0) {
    // max fragment usage failed
}
```

C.58.4.389 function wolfSSL_UseTruncatedHMAC

```
int wolfSSL_UseTruncatedHMAC(
    WOLFSSL * ssl
)
```

This function is called on the client side to enable the use of Truncated HMAC in the SSL object passed in the 'ssl' parameter. It means that the Truncated HMAC extension will be sent on ClientHello by wolfSSL clients.

Parameters:

- **ssl** pointer to a SSL object, created with wolfSSL_new()

See:

- wolfSSL_new
- wolfSSL_CTX_UseMaxFragment

Return:

- SSL_SUCCESS upon success.
- BAD_FUNC_ARG is the error that will be returned in one of these cases: ssl is NULL
- MEMORY_E is the error returned when there is not enough memory.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;
WOLFSSL* ssl = 0;
ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // context creation failed
}
ssl = wolfSSL_new(ctx);
if (ssl == NULL) {
```

```
    // ssl creation failed
}
ret = wolfSSL_UseTruncatedHMAC(ssl);
if (ret != 0) {
    // truncated HMAC usage failed
}
```

C.58.4.390 function wolfSSL_CTX_UseTruncatedHMAC

```
int wolfSSL_CTX_UseTruncatedHMAC(
    WOLFSSL_CTX * ctx
)
```

This function is called on the client side to enable the use of Truncated HMAC for SSL objects created from the SSL context passed in the 'ctx' parameter. It means that the Truncated HMAC extension will be sent on ClientHello by wolfSSL clients.

Parameters:

- **ctx** pointer to a SSL context, created with wolfSSL_CTX_new().

See:

- wolfSSL_CTX_new
- wolfSSL_UseMaxFragment

Return:

- SSL_SUCCESS upon success.
- BAD_FUNC_ARG is the error that will be returned in one of these cases: ctx is NULL
- MEMORY_E is the error returned when there is not enough memory.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;
ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // context creation failed
}
ret = wolfSSL_CTX_UseTruncatedHMAC(ctx);
if (ret != 0) {
    // truncated HMAC usage failed
}
```

C.58.4.391 function wolfSSL_UseOCSPStapling

```
int wolfSSL_UseOCSPStapling(
    WOLFSSL * ssl,
    unsigned char status_type,
    unsigned char options
)
```

Stapling eliminates the need to contact the CA. Stapling lowers the cost of certificate revocation check presented in OCSP.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

- **status_type** a byte type that is passed through to TLSX_UseCertificateStatusRequest() and stored in the CertificateStatusRequest structure.
- **options** a byte type that is passed through to TLSX_UseCertificateStatusRequest() and stored in the CertificateStatusRequest structure.

See:

- TLSX_UseCertificateStatusRequest
- wolfSSL_CTX_UseOCSPStapling

Return:

- SSL_SUCCESS returned if TLSX_UseCertificateStatusRequest executes without error.
- MEMORY_E returned if there is an error with the allocation of memory.
- BAD_FUNC_ARG returned if there is an argument that has a NULL or otherwise unacceptable value passed into the function.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
…
if (wolfSSL_UseOCSPStapling(ssl, WOLFSSL_CSR2_OCSP,
WOLFSSL_CSR2_OCSP_USE_NONCE) != SSL_SUCCESS){
    // Failed case.
}
```

C.58.4.392 function wolfSSL_CTX_UseOCSPStapling

```
int wolfSSL_CTX_UseOCSPStapling(
    WOLFSSL_CTX * ctx,
    unsigned char status_type,
    unsigned char options
)
```

This function requests the certificate status during the handshake.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **status_type** a byte type that is passed through to TLSX_UseCertificateStatusRequest() and stored in the CertificateStatusRequest structure.
- **options** a byte type that is passed through to TLSX_UseCertificateStatusRequest() and stored in the CertificateStatusRequest structure.

See:

- wolfSSL_UseOCSPStaplingV2
- wolfSSL_UseOCSPStapling
- TLSX_UseCertificateStatusRequest

Return:

- SSL_SUCCESS returned if the function and subroutines execute without error.
- BAD_FUNC_ARG returned if the WOLFSSL_CTX structure is NULL or otherwise if a unpermitted value is passed to a subroutine.
- MEMORY_E returned if the function or subroutine failed to properly allocate memory.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
byte statusRequest = 0; // Initialize status request
```

```
…
switch(statusRequest){
    case WOLFSSL_CSR_OCSP:
        if(wolfSSL_CTX_UseOCSPStapling(ssl->ctx, WOLFSSL_CSR_OCSP,
WOLF_CSR_OCSP_USE_NONCE) != SSL_SUCCESS){
// UseCertificateStatusRequest failed
}
// Continue switch cases
```

C.58.4.393 function wolfSSL_UseOCSPStaplingV2

```
int wolfSSL_UseOCSPStaplingV2(
    WOLFSSL * ssl,
    unsigned char status_type,
    unsigned char options
)
```

The function sets the status type and options for OCSP.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **status_type** a byte type that loads the OCSP status type.
- **options** a byte type that holds the OCSP options, set in wolfSSL_SNI_SetOptions() and wolfSSL_CTX_SNI_SetOptions().

See:

- TLSX_UseCertificatStatusRequestV2
- wolfSSL_SNI_SetOptions
- wolfSSL_CTX_SNI_SetOptions

Return:

- SSL_SUCCESS - returned if the function and subroutines executed without error.
- MEMORY_E - returned if there was an allocation of memory error.
- BAD_FUNC_ARG - returned if a NULL or otherwise unaccepted argument was passed to the function or a subroutine.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if (wolfSSL_UseOCSPStaplingV2(ssl, WOLFSSL_CSR2_OCSP_MULTI, 0) != SSL_SUCCESS){
    // Did not execute properly. Failure case code block.
}
```

C.58.4.394 function wolfSSL_CTX_UseOCSPStaplingV2

```
int wolfSSL_CTX_UseOCSPStaplingV2(
    WOLFSSL_CTX * ctx,
    unsigned char status_type,
    unsigned char options
)
```

Creates and initializes the certificate status request for OCSP Stapling.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

- **status_type** a byte type that is located in the CertificatStatusRequest structure and must be either WOLFSSL_CSR2_OCSP or WOLFSSL_CSR2_OCSP_MULTI.
- **options** a byte type that will be held in CertificateStatusRequestItemV2 struct.

See:

- TLSX_UseCertificateStatusRequestV2
- wc_RNG_GenerateBlock
- TLSX_Push

Return:

- SSL_SUCCESS if the function and subroutines executed without error.
- BAD_FUNC_ARG returned if the WOLFSSL_CTX structure is NULL or if the side variable is not client side.
- MEMORY_E returned if the allocation of memory failed.

Example

```
WOLFSSL_CTX* ctx  = wolfSSL_CTX_new( protocol method );
byte status_type;
byte options;
...
if(wolfSSL_CTX_UseOCSPStaplingV2(ctx, status_type, options); != SSL_SUCCESS){
    // Failure case.
}
```

C.58.4.395 function wolfSSL_UseSupportedCurve

```
int wolfSSL_UseSupportedCurve(
    WOLFSSL * ssl,
    word16 name
)
```

This function is called on the client side to enable the use of Supported Elliptic Curves Extension in the SSL object passed in the 'ssl' parameter. It means that the supported curves enabled will be sent on ClientHello by wolfSSL clients. This function can be called more than one time to enable multiple curves.

Parameters:

- **ssl** pointer to a SSL object, created with wolfSSL_new().
- **name** indicates which curve will be supported for the session. The available options are: enum { WOLFSSL_ECC_SECP160R1 = 0x10, WOLFSSL_ECC_SECP192R1 = 0x13, WOLFSSL_ECC_SECP224R1 = 0x15, WOLFSSL_ECC_SECP256R1 = 0x17, WOLFSSL_ECC_SECP384R1 = 0x18, WOLFSSL_ECC_SECP521R1 = 0x19 };

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_UseSupportedCurve

Return:

- SSL_SUCCESS upon success.
- BAD_FUNC_ARG is the error that will be returned in one of these cases: ssl is NULL, name is a unknown value. (see below)
- MEMORY_E is the error returned when there is not enough memory.
- SSL_FAILURE will be returned when TLS is not compiled in.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;
WOLFSSL* ssl = 0;
ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // context creation failed
}
ssl = wolfSSL_new(ctx);
if (ssl == NULL) {
    // ssl creation failed
}
ret = wolfSSL_UseSupportedCurve(ssl, WOLFSSL_ECC_SECP256R1);
if (ret != 0) {
    // Elliptic Curve Extension usage failed
}
```

C.58.4.396 function wolfSSL_CTX_UseSupportedCurve

```
int wolfSSL_CTX_UseSupportedCurve(
    WOLFSSL_CTX * ctx,
    word16 name
)
```

This function is called on the client side to enable the use of Supported Elliptic Curves Extension for SSL objects created from the SSL context passed in the 'ctx' parameter. It means that the supported curves enabled will be sent on ClientHello by wolfSSL clients. This function can be called more than one time to enable multiple curves.

Parameters:

- **ctx** pointer to a SSL context, created with wolfSSL_CTX_new().
- **name** indicates which curve will be supported for the session. The available options are: enum { WOLFSSL_ECC_SECP160R1 = 0x10, WOLFSSL_ECC_SECP192R1 = 0x13, WOLFSSL_ECC_SECP224R1 = 0x15, WOLFSSL_ECC_SECP256R1 = 0x17, WOLFSSL_ECC_SECP384R1 = 0x18, WOLFSSL_ECC_SECP521R1 = 0x19 };

See:

- wolfSSL_CTX_new
- wolfSSL_UseSupportedCurve

Return:

- SSL_SUCCESS upon success.
- BAD_FUNC_ARG is the error that will be returned in one of these cases: ctx is NULL, name is a unknown value. (see below)
- MEMORY_E is the error returned when there is not enough memory.
- SSL_FAILURE will be returned when TLS is not compiled in.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;
ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // context creation failed
}
ret = wolfSSL_CTX_UseSupportedCurve(ctx, WOLFSSL_ECC_SECP256R1);
if (ret != 0) {
```

```
    // Elliptic Curve Extension usage failed
}
```

C.58.4.397 function wolfSSL_UseSecureRenegotiation

```
int wolfSSL_UseSecureRenegotiation(
    WOLFSSL * ssl
)
```

This function forces secure renegotiation for the supplied WOLFSSL structure. This is not recommended.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- TLSX_Find
- TLSX_UseSecureRenegotiation

Return:

- SSL_SUCCESS Successfully set secure renegotiation.
- BAD_FUNC_ARG Returns error if ssl is null.
- MEMORY_E Returns error if unable to allocate memory for secure renegotiation.

Example

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
WOLFSSL_METHOD method = // Some wolfSSL method
ctx = wolfSSL_CTX_new(method);
ssl = wolfSSL_new(ctx);

if(wolfSSL_UseSecureRenegotiation(ssl) != SSL_SUCCESS)
{
    // Error setting secure renegotiation
}
```

C.58.4.398 function wolfSSL_Rehandshake

```
int wolfSSL_Rehandshake(
    WOLFSSL * ssl
)
```

This function executes a secure renegotiation handshake; this is user forced as wolfSSL discourages this functionality.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_negotiate
- wc_InitSha512
- wc_InitSha384
- wc_InitSha256
- wc_InitSha

- wc_InitMd5

Return:

- SSL_SUCCESS returned if the function executed without error.
- BAD_FUNC_ARG returned if an unacceptable argument was passed in a subroutine.
- SECURE_RENEGOTIATION_E returned if there was an error with renegotiating the handshake.
- SSL_FATAL_ERROR returned if there was an error with the server or client configuration and the renegotiation could not be completed. See wolfSSL_negotiate().
- SSL_FAILURE will be returned when ssl is NULL.

If decrypted application data is waiting to be read, the renegotiation is not continued and wolfSSL_get_error() returns APP_DATA_READY. Read the data with wolfSSL_read(), which also moves the renegotiation forward, or call wolfSSL_Rehandshake() again once it has been read.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_Rehandshake(ssl) != SSL_SUCCESS){
    // There was an error and the rehandshake is not successful.
}
```

C.58.4.399 function wolfSSL_UseSessionTicket

```
int wolfSSL_UseSessionTicket(
    WOLFSSL * ssl
)
```

Force provided WOLFSSL structure to use session ticket. The constant HAVE_SESSION_TICKET should be defined and the constant NO_WOLFSSL_CLIENT should not be defined to use this function.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: TLSX_UseSessionTicket

Return:

- SSL_SUCCESS Successfully set use session ticket.
- BAD_FUNC_ARG Returned if ssl is null.
- MEMORY_E Error allocating memory for setting session ticket.

Example

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
WOLFSSL_METHOD method = // Some wolfSSL method
ctx = wolfSSL_CTX_new(method);
ssl = wolfSSL_new(ctx);

if(wolfSSL_UseSessionTicket(ssl) != SSL_SUCCESS)
{
    // Error setting session ticket
}
```

C.58.4.400 function wolfSSL_CTX_UseSessionTicket

```
int wolfSSL_CTX_UseSessionTicket(
    WOLFSSL_CTX * ctx
)
```

This function sets wolfSSL context to use a session ticket.

Parameters:

- **ctx** The WOLFSSL_CTX structure to use.

See: TLSX_UseSessionTicket

Return:

- SSL_SUCCESS Function executed successfully.
- BAD_FUNC_ARG Returned if ctx is null.
- MEMORY_E Error allocating memory in internal function.

Example

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL_METHOD method = // Some wolfSSL method ;
ctx = wolfSSL_CTX_new(method);

if(wolfSSL_CTX_UseSessionTicket(ctx) != SSL_SUCCESS)
{
    // Error setting session ticket
}
```

C.58.4.401 function wolfSSL_get_SessionTicket

```
int wolfSSL_get_SessionTicket(
    WOLFSSL * ssl,
    unsigned char * buf,
    word32 * bufSz
)
```

This function copies the ticket member of the Session structure to the buffer. If buf is NULL and bufSz is non-NULL, bufSz will be set to the ticket length.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **buf** a byte pointer representing the memory buffer.
- **bufSz** a word32 pointer representing the buffer size.

See:

- wolfSSL_UseSessionTicket
- wolfSSL_set_SessionTicket

Return:

- SSL_SUCCESS returned if the function executed without error.
- BAD_FUNC_ARG returned if ssl or bufSz is NULL, or if bufSz is non-NULL and buf is NULL
- LENGTH_ONLY_E will be returned when buf is NULL, with the required length returned through bufSz.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
byte* buf;
word32 bufSz;  // Initialize with buf size
…
if(wolfSSL_get_SessionTicket(ssl, buf, bufSz) <= 0){
    // Nothing was written to the buffer
} else {
    // the buffer holds the content from ssl->session->ticket
}
```

C.58.4.402 function wolfSSL_set_SessionTicket

```
int wolfSSL_set_SessionTicket(
    WOLFSSL * ssl,
    const unsigned char * buf,
    word32 bufSz
)
```

This function sets the ticket member of the WOLFSSL_SESSION structure within the WOLFSSL struct. The buffer passed into the function is copied to memory.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **buf** a byte pointer that gets loaded into the ticket member of the session structure.
- **bufSz** a word32 type that represents the size of the buffer.

See: wolfSSL_set_SessionTicket_cb

Return:

- SSL_SUCCESS returned on successful execution of the function. The function returned without errors.
- BAD_FUNC_ARG returned if the WOLFSSL structure is NULL. This will also be thrown if the buf argument is NULL but the bufSz argument is not zero.
- MEMORY_ERROR will be returned when the ticket cannot be allocated.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
byte* buffer; // File to load
word32 bufSz;
...
if(wolfSSL_KeepArrays(ssl, buffer, bufSz) != SSL_SUCCESS){
    // There was an error loading the buffer to memory.
}
```

C.58.4.403 function wolfSSL_set_SessionTicket_cb

```
int wolfSSL_set_SessionTicket_cb(
    WOLFSSL * ssl,
    CallbackSessionTicket cb,
    void * ctx
)
```

This function sets the session ticket callback. The type CallbackSessionTicket is a function pointer with the signature of: int (*CallbackSessionTicket)(WOLFSSL*, const unsigned char*, int, void*)

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **cb** a function pointer to the type CallbackSessionTicket.
- **ctx** a void pointer to the session_ticket_ctx member of the WOLFSSL structure.

See:

- wolfSSL_get_SessionTicket
- CallbackSessionTicket
- sessionTicketCB

Return:

- SSL_SUCCESS returned if the function executed without error.
- BAD_FUNC_ARG returned if the WOLFSSL structure is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
…
int sessionTicketCB(WOLFSSL* ssl, const unsigned char* ticket, int ticketSz,
            void* ctx){ … }
wolfSSL_set_SessionTicket_cb(ssl, sessionTicketCB, (void*)"initial session");
```

C.58.4.404 function wolfSSL_send_SessionTicket

```
int wolfSSL_send_SessionTicket(
    WOLFSSL * ssl
)
```

This function sends a session ticket to the client after a TLS v1.3 handhsake has been established.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_get_SessionTicket
- CallbackSessionTicket
- sessionTicketCB

Return:

- WOLFSSL_SUCCESS returned if a new session ticket was sent.
- BAD_FUNC_ARG returned if WOLFSSL structure is NULL, or not using TLS v1.3.
- SIDE_ERROR returned if not a server.
- NOT_READY_ERROR returned if the handshake has not completed.
- WOLFSSL_FATAL_ERROR returned if creating or sending message fails.

Example

```
int ret;
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
…
ret = wolfSSL_send_SessionTicket(ssl);
if (ret != WOLFSSL_SUCCESS) {
    // New session ticket not sent.
}
```

C.58.4.405 function wolfSSL_CTX_set_TicketEncCb

```
int wolfSSL_CTX_set_TicketEncCb(
    WOLFSSL_CTX * ctx,
    SessionTicketEncCb cb
)
```

This function sets the session ticket key encrypt callback function for a server to support session tickets as specified in RFC 5077.

Parameters:

- **ctx** pointer to the WOLFSSL_CTX object, created with wolfSSL_CTX_new().
- **cb** user callback function to encrypt/decrypt session tickets
- **ssl(Callback)** pointer to the WOLFSSL object, created with wolfSSL_new()
- **key_name(Callback)** unique key name for this ticket context, should be randomly generated
- **iv(Callback)** unique IV for this ticket, up to 128 bits, should be randomly generated
- **mac(Callback)** up to 256 bit mac for this ticket
- **enc(Callback)** if this encrypt parameter is true the user should fill in key_name, iv, mac, and encrypt the ticket in-place of length inLen and set the resulting output length in *outLen. Returning WOLFSSL_TICKET_RET_OK tells wolfSSL that the encryption was successful. If this encrypt parameter is false, the user should perform a decrypt of the ticket in-place of length inLen using key_name, iv, and mac. The resulting decrypt length should be set in* outLen. Returning WOLFSSL_TICKET_RET_OK tells wolfSSL to proceed using the decrypted ticket. Returning WOLFSSL_TICKET_RET_CREATE tells wolfSSL to use the decrypted ticket but also to generate a new one to send to the client, helpful if recently rolled keys and don't want to force a full handshake. Returning WOLFSSL_TICKET_RET_REJECT tells wolfSSL to reject this ticket, perform a full handshake, and create a new standard session ID for normal session resumption. Returning WOLFSSL_TICKET_RET_FATAL tells wolfSSL to end the connection attempt with a fatal error.
- **ticket(Callback)** the input/output buffer for the encrypted ticket. See the enc parameter
- **inLen(Callback)** the input length of the ticket parameter
- **outLen(Callback)** the resulting output length of the ticket parameter. When entering the callback outLen will indicate the maximum size available in the ticket buffer.
- **userCtx(Callback)** the user context set with wolfSSL_CTX_set_TicketEncCtx()

See:

- wolfSSL_CTX_set_TicketHint
- wolfSSL_CTX_set_TicketEncCtx

Return:

- SSL_SUCCESS will be returned upon successfully setting the session.
- BAD_FUNC_ARG will be returned on failure. This is caused by passing invalid arguments to the function.

Example

```
See wolfssl/test.h myTicketEncCb() used by the example
server and example echoserver.
```

C.58.4.406 function wolfSSL_CTX_set_TicketHint

```
int wolfSSL_CTX_set_TicketHint(
    WOLFSSL_CTX * ctx,
    int hint
)
```

This function sets the session ticket hint relayed to the client. For server side use.

Parameters:

- **ctx** pointer to the WOLFSSL_CTX object, created with wolfSSL_CTX_new().
- **hint** number of seconds the ticket might be valid for. Hint to client.

See: wolfSSL_CTX_set_TicketEncCb

Return:

- SSL_SUCCESS will be returned upon successfully setting the session.
- BAD_FUNC_ARG will be returned on failure. This is caused by passing invalid arguments to the function.

Example

none

C.58.4.407 function wolfSSL_CTX_set_TicketEncCtx

```
int wolfSSL_CTX_set_TicketEncCtx(
    WOLFSSL_CTX * ctx,
    void * userCtx
)
```

This function sets the session ticket encrypt user context for the callback. For server side use.

Parameters:

- **ctx** pointer to the WOLFSSL_CTX object, created with wolfSSL_CTX_new().
- **userCtx** the user context for the callback

See: wolfSSL_CTX_set_TicketEncCb

Return:

- SSL_SUCCESS will be returned upon successfully setting the session.
- BAD_FUNC_ARG will be returned on failure. This is caused by passing invalid arguments to the function.

Example

none

C.58.4.408 function wolfSSL_CTX_get_TicketEncCtx

```
void * wolfSSL_CTX_get_TicketEncCtx(
    WOLFSSL_CTX * ctx
)
```

This function gets the session ticket encrypt user context for the callback. For server side use.

Parameters:

- **ctx** pointer to the WOLFSSL_CTX object, created with wolfSSL_CTX_new().

See: wolfSSL_CTX_set_TicketEncCtx

Return:

- userCtx will be returned upon successfully getting the session.
- NULL will be returned on failure. This is caused by passing invalid arguments to the function, or when the user context has not been set.

Example

none

C.58.4.409 function wolfSSL_SetHsDoneCb

```
int wolfSSL_SetHsDoneCb(
    WOLFSSL * ssl,
    HandShakeDoneCb cb,
    void * user_ctx
)
```

This function sets the handshake done callback. The hsDoneCb and hsDoneCtx members of the WOLFSSL structure are set in this function.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **cb** a function pointer of type HandShakeDoneCb with the signature of the form: int (*HandShakeDoneCb)(WOLFSSL*, void*);
- **user_ctx** a void pointer to the user registered context.

See: HandShakeDoneCb

Return:

- SSL_SUCCESS returned if the function executed without an error. The hsDoneCb and hsDoneCtx members of the WOLFSSL struct are set.
- BAD_FUNC_ARG returned if the WOLFSSL struct is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
…
int myHsDoneCb(WOLFSSL* ssl, void* user_ctx){
    // callback function
}
…
wolfSSL_SetHsDoneCb(ssl, myHsDoneCb, NULL);
```

C.58.4.410 function wolfSSL_PrintSessionStats

```
int wolfSSL_PrintSessionStats(
    void
)
```

This function prints the statistics from the session.

Parameters:

- **none** No parameters.

See: wolfSSL_get_session_stats

Return:

- SSL_SUCCESS returned if the function and subroutines return without error. The session stats have been successfully retrieved and printed.
- BAD_FUNC_ARG returned if the subroutine wolfSSL_get_session_stats() was passed an unacceptable argument.
- BAD_MUTEX_E returned if there was a mutex error in the subroutine.

Example

```
// You will need to have a session object to retrieve stats from.
if(wolfSSL_PrintSessionStats(void) != SSL_SUCCESS   ){
    // Did not print session stats
}
```

C.58.4.411 function wolfSSL_get_session_stats

```
int wolfSSL_get_session_stats(
    unsigned int * active,
    unsigned int * total,
    unsigned int * peak,
    unsigned int * maxSessions
)
```

This function gets the statistics for the session.

Parameters:

- **active** a word32 pointer representing the total current sessions.
- **total** a word32 pointer representing the total sessions.
- **peak** a word32 pointer representing the peak sessions.
- **maxSessions** a word32 pointer representing the maximum sessions.

See: wolfSSL_PrintSessionStats

Return:

- SSL_SUCCESS returned if the function and subroutines return without error. The session stats have been successfully retrieved and printed.
- BAD_FUNC_ARG returned if the subroutine wolfSSL_get_session_stats() was passed an unacceptable argument.
- BAD_MUTEX_E returned if there was a mutex error in the subroutine.

Example

```
int wolfSSL_PrintSessionStats(void){
…
ret = wolfSSL_get_session_stats(&totalSessionsNow,
&totalSessionsSeen, &peak, &maxSessions);
…
return ret;
```

C.58.4.412 function wolfSSL_MakeTlsMasterSecret

```
int wolfSSL_MakeTlsMasterSecret(
    unsigned char * ms,
    word32 msLen,
    const unsigned char * pms,
    word32 pmsLen,
    const unsigned char * cr,
    const unsigned char * sr,
    int tls1_2,
    int hash_type
)
```

This function copies the values of cr and sr then passes through to wc_PRF (pseudo random function) and returns that value.

Parameters:

- **ms** the master secret held in the Arrays structure.
- **msLen** the length of the master secret.
- **pms** the pre-master secret held in the Arrays structure.
- **pmsLen** the length of the pre-master secret.
- **cr** the client random.
- **sr** the server random.
- **tls1_2** signifies that the version is at least tls version 1.2.
- **hash_type** signifies the hash type.

See:

- wc_PRF
- MakeTlsMasterSecret

Return:

- 0 on success
- BUFFER_E returned if there will be an error with the size of the buffer.
- MEMORY_E returned if a subroutine failed to allocate dynamic memory.

Example

```
WOLFSSL* ssl;

called in MakeTlsMasterSecret and retrieves the necessary
information as follows:

int MakeTlsMasterSecret(WOLFSSL* ssl){
int ret;
ret = wolfSSL_makeTlsMasterSecret(ssl->arrays->masterSecret, SECRET_LEN,
ssl->arrays->preMasterSecret, ssl->arrays->preMasterSz,
ssl->arrays->clientRandom, ssl->arrays->serverRandom,
IsAtLeastTLSv1_2(ssl), ssl->specs.mac_algorithm);
…
return ret;

}
```

C.58.4.413 function wolfSSL_DeriveTlsKeys

```
int wolfSSL_DeriveTlsKeys(
    unsigned char * key_data,
    word32 keyLen,
    const unsigned char * ms,
    word32 msLen,
    const unsigned char * sr,
    const unsigned char * cr,
    int tls1_2,
    int hash_type
)
```

An external facing wrapper to derive TLS Keys.

Parameters:

- **key_data** a byte pointer that is allocateded in DeriveTlsKeys and passed through to wc_PRF to hold the final hash.
- **keyLen** a word32 type that is derived in DeriveTlsKeys from the WOLFSSL structure's specs member.

- **ms** a constant pointer type holding the master secret held in the arrays structure within the WOLFSSL structure.
- **msLen** a word32 type that holds the length of the master secret in an enumerated define, SECRET_LEN.
- **sr** a constant byte pointer to the serverRandom member of the arrays structure within the WOLFSSL structure.
- **cr** a constant byte pointer to the clientRandom member of the arrays structure within the WOLFSSL structure.
- **tls1_2** an integer type returned from IsAtLeastTLSv1_2().
- **hash_type** an integer type held in the WOLFSSL structure.

See:

- wc_PRF
- DeriveTlsKeys
- IsAtLeastTLSv1_2

Return:

- 0 returned on success.
- BUFFER_E returned if the sum of labLen and seedLen (computes total size) exceeds the maximum size.
- MEMORY_E returned if the allocation of memory failed.

Example

```
int DeriveTlsKeys(WOLFSSL* ssl){
int ret;
…
ret = wolfSSL_DeriveTlsKeys(key_data, length, ssl->arrays->masterSecret,
SECRET_LEN, ssl->arrays->clientRandom,
IsAtLeastTLSv1_2(ssl), ssl->specs.mac_algorithm);
…
}
```

C.58.4.414 function wolfSSL_connect_ex

```
int wolfSSL_connect_ex(
    WOLFSSL * ssl,
    HandShakeCallBack hsCb,
    TimeoutCallBack toCb,
    WOLFSSL_TIMEVAL timeout
)
```

wolfSSL_connect_ex() is an extension that allows a HandShake Callback to be set. This can be useful in embedded systems for debugging support when a debugger isn't available and sniffing is impractical. The HandShake Callback will be called whether or not a handshake error occurred. No dynamic memory is used since the maximum number of SSL packets is known. Packet names can be accessed through packetNames[]. The connect extension also allows a Timeout Callback to be set along with a timeout value. This is useful if the user doesn't want to wait for the TCP stack to timeout. This extension can be called with either, both, or neither callbacks.

Parameters:

- **ssl** pointer to a WOLFSSL structure, created using wolfSSL_new().
- **hsCb** HandShake Callback function pointer.
- **toCb** Timeout Callback function pointer.
- **timeout** timeout value to use with the Timeout Callback.

See: wolfSSL_accept_ex

Return:

- SSL_SUCCESS upon success.
- GETTIME_ERROR will be returned if gettimeofday() encountered an error.
- SETITIMER_ERROR will be returned if setitimer() encountered an error.
- SIGACT_ERROR will be returned if sigaction() encountered an error.
- SSL_FATAL_ERROR will be returned if the underlying SSL_connect() call encountered an error.

Example

none

C.58.4.415 function wolfSSL_accept_ex

```
int wolfSSL_accept_ex(
    WOLFSSL * ssl,
    HandShakeCallBack hsCb,
    TimeoutCallBack toCb,
    WOLFSSL_TIMEVAL timeout
)
```

wolfSSL_accept_ex() is an extension that allows a HandShake Callback to be set. This can be useful in embedded systems for debugging support when a debugger isn't available and sniffing is impractical. The HandShake Callback will be called whether or not a handshake error occurred. No dynamic memory is used since the maximum number of SSL packets is known. Packet names can be accessed through packetNames[]. The connect extension also allows a Timeout Callback to be set along with a timeout value. This is useful if the user doesn't want to wait for the TCP stack to timeout. This extension can be called with either, both, or neither callbacks.

Parameters:

- **ssl** pointer to a WOLFSSL structure, created using wolfSSL_new().
- **hsCb** HandShake Callback function pointer.
- **toCb** Timeout Callback function pointer.
- **timeout** timeout value to use with the Timeout Callback.

See: wolfSSL_connect_ex

Return:

- SSL_SUCCESS upon success.
- GETTIME_ERROR will be returned if gettimeofday() encountered an error.
- SETITIMER_ERROR will be returned if setitimer() encountered an error.
- SIGACT_ERROR will be returned if sigaction() encountered an error.
- SSL_FATAL_ERROR will be returned if the underlying SSL_accept() call encountered an error.

Example

none

C.58.4.416 function wolfSSL_BIO_set_fp

```
long wolfSSL_BIO_set_fp(
    WOLFSSL_BIO * bio,
    XFILE fp,
    int c
)
```

This is used to set the internal file pointer for a BIO.

Parameters:

- **bio** WOLFSSL_BIO structure to set pair.
- **fp** file pointer to set in bio.
- **c** close file behavior flag.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_get_fp
- wolfSSL_BIO_free

Return:

- SSL_SUCCESS On successfully setting file pointer.
- SSL_FAILURE If an error case was encountered.

Example

```
WOLFSSL_BIO* bio;
XFILE fp;
int ret;
bio  = wolfSSL_BIO_new(wolfSSL_BIO_s_file());
ret  = wolfSSL_BIO_set_fp(bio, fp, BIO_CLOSE);
// check ret value
```

C.58.4.417 function wolfSSL_BIO_get_fp

```
long wolfSSL_BIO_get_fp(
    WOLFSSL_BIO * bio,
    XFILE * fp
)
```

This is used to get the internal file pointer for a BIO.

Parameters:

- **bio** WOLFSSL_BIO structure to set pair.
- **fp** file pointer to set in bio.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_set_fp
- wolfSSL_BIO_free

Return:

- SSL_SUCCESS On successfully getting file pointer.
- SSL_FAILURE If an error case was encountered.

Example

```
WOLFSSL_BIO* bio;
XFILE fp;
int ret;
bio  = wolfSSL_BIO_new(wolfSSL_BIO_s_file());
ret  = wolfSSL_BIO_get_fp(bio, &fp);
// check ret value
```

C.58.4.418 function wolfSSL_check_private_key

```
int wolfSSL_check_private_key(
    const WOLFSSL * ssl
)
```

This function checks that the private key is a match with the certificate being used.

Parameters:

- **ssl** WOLFSSL structure to check.

See:

- wolfSSL_new
- wolfSSL_free

Return:

- SSL_SUCCESS On successfully match.
- SSL_FAILURE If an error case was encountered.
- <0 All error cases other than SSL_FAILURE are negative values.

Example

```
WOLFSSL* ssl;
int ret;
// create and set up ssl
ret  = wolfSSL_check_private_key(ssl);
// check ret value
```

C.58.4.419 function wolfSSL_X509_get_ext_by_NID

```
int wolfSSL_X509_get_ext_by_NID(
    const WOLFSSL_X509 * x,
    int nid,
    int lastpos
)
```

This function looks for and returns the extension index matching the passed in NID value.

Parameters:

- **x** certificate to get parse through for extension.
- **nid** extension OID to be found.
- **lastpos** start search from extension after lastpos. Set to -1 initially.

Return:

- = 0 If successful the extension index is returned.
- -1 If extension is not found or error is encountered.

Example

```
const WOLFSSL_X509* x509;
int lastPos = -1;
int idx;

idx = wolfSSL_X509_get_ext_by_NID(x509, NID_basic_constraints, lastPos);
```

C.58.4.420 function wolfSSL_X509_get_ext_d2i

```
void * wolfSSL_X509_get_ext_d2i(
    const WOLFSSL_X509 * x509,
    int nid,
    int * c,
    int * idx
)
```

This function looks for and returns the extension matching the passed in NID value.

Parameters:

- **x509** certificate to get parse through for extension.
- **nid** extension OID to be found.
- **c** if not NULL is set to -2 for multiple extensions found -1 if not found, 0 if found and not critical and 1 if found and critical.
- **idx** if NULL return first extension matched otherwise if not stored in x509 start at idx.

See: wolfSSL_sk_ASN1_OBJECT_free

Return:

- pointer If successful a STACK_OF(WOLFSSL_ASN1_OBJECT) pointer is returned.
- NULL If extension is not found or error is encountered.

Example

```
const WOLFSSL_X509* x509;
int c;
int idx = 0;
STACK_OF(WOLFSSL_ASN1_OBJECT)* sk;

sk = wolfSSL_X509_get_ext_d2i(x509, NID_basic_constraints, &c, &idx);
//check sk for NULL and then use it. sk needs freed after done.
```

C.58.4.421 function wolfSSL_X509_digest

```
int wolfSSL_X509_digest(
    const WOLFSSL_X509 * x509,
    const WOLFSSL_EVP_MD * digest,
    unsigned char * buf,
    unsigned int * len
)
```

This function returns the hash of the DER certificate.

Parameters:

- **x509** certificate to get the hash of.
- **digest** the hash algorithm to use.
- **buf** buffer to hold hash.
- **len** length of buffer.

See: none

Return:

- SSL_SUCCESS On successfully creating a hash.
- SSL_FAILURE Returned on bad input or unsuccessful hash.

Example

```
WOLFSSL_X509* x509;
unsigned char buffer[64];
unsigned int bufferSz;
int ret;

ret = wolfSSL_X509_digest(x509, wolfSSL_EVP_sha256(), buffer, &bufferSz);
//check ret value
```

C.58.4.422 function wolfSSL_use_certificate

```
int wolfSSL_use_certificate(
    WOLFSSL * ssl,
    WOLFSSL_X509 * x509
)
```

his is used to set the certificate for WOLFSSL structure to use during a handshake.

Parameters:

- **ssl** WOLFSSL structure to set certificate in.
- **x509** certificate to use.

See:

- wolfSSL_new
- wolfSSL_free

Return:

- SSL_SUCCESS On successful setting argument.
- SSL_FAILURE If a NULL argument passed in.

Example

```
 WOLFSSL* ssl;
WOLFSSL_X509* x509
int ret;
// create ssl object and x509
ret  = wolfSSL_use_certificate(ssl, x509);
// check ret value
```

C.58.4.423 function wolfSSL_use_certificate_ASN1

```
int wolfSSL_use_certificate_ASN1(
    WOLFSSL * ssl,
    const unsigned char * der,
    int derSz
)
```

This is used to set the certificate for WOLFSSL structure to use during a handshake. A DER formatted buffer is expected.

Parameters:

- **ssl** WOLFSSL structure to set certificate in.
- **der** DER certificate to use.
- **derSz** size of the DER buffer passed in.

See:

- wolfSSL_new

- wolfSSL_free

Return:

- SSL_SUCCESS On successful setting argument.
- SSL_FAILURE If a NULL argument passed in.

Example

```
WOLFSSL* ssl;
unsigned char* der;
int derSz;
int ret;
// create ssl object and set DER variables
ret  = wolfSSL_use_certificate_ASN1(ssl, der, derSz);
// check ret value
```

C.58.4.424 function wolfSSL_use_PrivateKey

```
int wolfSSL_use_PrivateKey(
    WOLFSSL * ssl,
    WOLFSSL_EVP_PKEY * pkey
)
```

This is used to set the private key for the WOLFSSL structure.

Parameters:

- **ssl** WOLFSSL structure to set argument in.
- **pkey** private key to use.

See:

- wolfSSL_new
- wolfSSL_free

Return:

- SSL_SUCCESS On successful setting argument.
- SSL_FAILURE If a NULL ssl passed in. All error cases will be negative values.

Example

```
WOLFSSL* ssl;
WOLFSSL_EVP_PKEY* pkey;
int ret;
// create ssl object and set up private key
ret  = wolfSSL_use_PrivateKey(ssl, pkey);
// check ret value
```

C.58.4.425 function wolfSSL_use_PrivateKey_ASN1

```
int wolfSSL_use_PrivateKey_ASN1(
    int pri,
    WOLFSSL * ssl,
    const unsigned char * der,
    long derSz
)
```

This is used to set the private key for the WOLFSSL structure. A DER formatted key buffer is expected.

Parameters:

- **pri** type of private key.
- **ssl** WOLFSSL structure to set argument in.
- **der** buffer holding DER key.
- **derSz** size of der buffer.

See:

- wolfSSL_new
- wolfSSL_free
- wolfSSL_use_PrivateKey

Return:

- SSL_SUCCESS On successful setting parsing and setting the private key.
- SSL_FAILURE If an NULL ssl passed in. All error cases will be negative values.

Example

```
WOLFSSL* ssl;
unsigned char* pkey;
long pkeySz;
int ret;
// create ssl object and set up private key
ret  = wolfSSL_use_PrivateKey_ASN1(1, ssl, pkey, pkeySz);
// check ret value
```

C.58.4.426 function wolfSSL_use_RSAPrivateKey_ASN1

```
int wolfSSL_use_RSAPrivateKey_ASN1(
    WOLFSSL * ssl,
    unsigned char * der,
    long derSz
)
```

This is used to set the private key for the WOLFSSL structure. A DER formatted RSA key buffer is expected.

Parameters:

- **ssl** WOLFSSL structure to set argument in.
- **der** buffer holding DER key.
- **derSz** size of der buffer.

See:

- wolfSSL_new
- wolfSSL_free
- wolfSSL_use_PrivateKey

Return:

- SSL_SUCCESS On successful setting parsing and setting the private key.
- SSL_FAILURE If an NULL ssl passed in. All error cases will be negative values.

Example

```
WOLFSSL* ssl;
unsigned char* pkey;
long pkeySz;
int ret;
// create ssl object and set up RSA private key
```

```
ret  = wolfSSL_use_RSAPrivateKey_ASN1(ssl, pkey, pkeySz);
// check ret value
```

C.58.4.427 function wolfSSL_DSA_dup_DH

```
WOLFSSL_DH * wolfSSL_DSA_dup_DH(
    const WOLFSSL_DSA * r
)
```

This function duplicates the parameters in dsa to a newly created WOLFSSL_DH structure.

Parameters:

- **dsa** WOLFSSL_DSA structure to duplicate.

See: none

Return:

- WOLFSSL_DH If duplicated returns WOLFSSL_DH structure
- NULL upon failure

Example

```
WOLFSSL_DH* dh;
WOLFSSL_DSA* dsa;
// set up dsa
dh = wolfSSL_DSA_dup_DH(dsa);

// check dh is not null
```

C.58.4.428 function wolfSSL_SESSION_get_master_key

```
int wolfSSL_SESSION_get_master_key(
    const WOLFSSL_SESSION * ses,
    unsigned char * out,
    int outSz
)
```

This is used to get the master key after completing a handshake.

Parameters:

- **ses** WOLFSSL_SESSION structure to get master secret buffer from.
- **out** buffer to hold data.
- **outSz** size of out buffer passed in. (if 0 function will return max buffer size needed)

See:

- wolfSSL_new
- wolfSSL_free

Return:

- 0 On successfully getting data returns a value greater than 0
- 0 If no random data buffer or an error state returns 0
- max If outSz passed in is 0 then the maximum buffer size needed is returned

Example

```
WOLFSSL_SESSION ssl;
unsigned char* buffer;
size_t bufferSz;
size_t ret;
// complete handshake and get session structure
bufferSz  = wolfSSL_SESSION_get_master_secret(ses, NULL, 0);
buffer = malloc(bufferSz);
ret  = wolfSSL_SESSION_get_master_secret(ses, buffer, bufferSz);
// check ret value
```

C.58.4.429 function wolfSSL_SESSION_get_master_key_length

```
int wolfSSL_SESSION_get_master_key_length(
    const WOLFSSL_SESSION * ses
)
```

This is used to get the master secret key length.

Parameters:

- **ses** WOLFSSL_SESSION structure to get master secret buffer from.

See:

- wolfSSL_new
- wolfSSL_free

Return: size Returns master secret key size.

Example

```
WOLFSSL_SESSION ssl;
unsigned char* buffer;
size_t bufferSz;
size_t ret;
// complete handshake and get session structure
bufferSz  = wolfSSL_SESSION_get_master_secret_length(ses);
buffer = malloc(bufferSz);
// check ret value
```

C.58.4.430 function wolfSSL_CTX_set_cert_store

```
void wolfSSL_CTX_set_cert_store(
    WOLFSSL_CTX * ctx,
    WOLFSSL_X509_STORE * str
)
```

This is a setter function for the WOLFSSL_X509_STORE structure in ctx.

Parameters:

- **ctx** pointer to the WOLFSSL_CTX structure for setting cert store pointer.
- **str** pointer to the WOLFSSL_X509_STORE to set in ctx.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free

Return: none No return.

Example

```
WOLFSSL_CTX ctx;
WOLFSSL_X509_STORE* st;
// setup ctx and st
st = wolfSSL_CTX_set_cert_store(ctx, st);
//use st
```

C.58.4.431 function wolfSSL_d2i_X509_bio

```
WOLFSSL_X509 * wolfSSL_d2i_X509_bio(
    WOLFSSL_BIO * bio,
    WOLFSSL_X509 ** x509
)
```

This function get the DER buffer from bio and converts it to a WOLFSSL_X509 structure.

Parameters:

- **bio** pointer to the WOLFSSL_BIO structure that has the DER certificate buffer.
- **x509** pointer that get set to new WOLFSSL_X509 structure created.

See: none

Return:

- pointer returns a WOLFSSL_X509 structure pointer on success.
- Null returns NULL on failure

Example

```
WOLFSSL_BIO* bio;
WOLFSSL_X509* x509;
// load DER into bio
x509 = wolfSSL_d2i_X509_bio(bio, NULL);
Or
wolfSSL_d2i_X509_bio(bio, &x509);
// use x509 returned (check for NULL)
```

C.58.4.432 function wolfSSL_CTX_get_cert_store

```
WOLFSSL_X509_STORE * wolfSSL_CTX_get_cert_store(
    WOLFSSL_CTX * ctx
)
```

This is a getter function for the WOLFSSL_X509_STORE structure in ctx.

Parameters:

- **ctx** pointer to the WOLFSSL_CTX structure for getting cert store pointer.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free
- wolfSSL_CTX_set_cert_store

Return:

- WOLFSSL_X509_STORE* On successfully getting the pointer.
- NULL Returned if NULL arguments are passed in.

Example

```
WOLFSSL_CTX ctx;
WOLFSSL_X509_STORE* st;
// setup ctx
st = wolfSSL_CTX_get_cert_store(ctx);
//use st
```

C.58.4.433 function wolfSSL_BIO_ctrl_pending

```
size_t wolfSSL_BIO_ctrl_pending(
    WOLFSSL_BIO * b
)
```

Gets the number of pending bytes to read. If BIO type is BIO_BIO then is the number to read from pair. If BIO contains an SSL object then is pending data from SSL object (wolfSSL_pending(ssl)). If is BIO_MEMORY type then returns the size of memory buffer.

Parameters:

- **bio** pointer to the WOLFSSL_BIO structure that has already been created.

See:

- wolfSSL_BIO_make_bio_pair
- wolfSSL_BIO_new

Return: >=0 number of pending bytes.

Example

```
WOLFSSL_BIO* bio;
int pending;
bio = wolfSSL_BIO_new();
…
pending = wolfSSL_BIO_ctrl_pending(bio);
```

C.58.4.434 function wolfSSL_get_server_random

```
size_t wolfSSL_get_server_random(
    const WOLFSSL * ssl,
    unsigned char * out,
    size_t outlen
)
```

This is used to get the random data sent by the server during the handshake.

Parameters:

- **ssl** WOLFSSL structure to get clients random data buffer from.
- **out** buffer to hold random data.
- **outSz** size of out buffer passed in. (if 0 function will return max buffer size needed)

See:

- wolfSSL_new
- wolfSSL_free

Return:

- 0 On successfully getting data returns a value greater than 0
- 0 If no random data buffer or an error state returns 0
- max If outSz passed in is 0 then the maximum buffer size needed is returned

Example

```
WOLFSSL ssl;
unsigned char* buffer;
size_t bufferSz;
size_t ret;
bufferSz  = wolfSSL_get_server_random(ssl, NULL, 0);
buffer = malloc(bufferSz);
ret  = wolfSSL_get_server_random(ssl, buffer, bufferSz);
// check ret value
```

C.58.4.435 function wolfSSL_get_client_random

```
size_t wolfSSL_get_client_random(
    const WOLFSSL * ssl,
    unsigned char * out,
    size_t outSz
)
```

This is used to get the random data sent by the client during the handshake.

Parameters:

- **ssl** WOLFSSL structure to get clients random data buffer from.
- **out** buffer to hold random data.
- **outSz** size of out buffer passed in. (if 0 function will return max buffer size needed)

See:

- wolfSSL_new
- wolfSSL_free

Return:

- 0 On successfully getting data returns a value greater than 0
- 0 If no random data buffer or an error state returns 0
- max If outSz passed in is 0 then the maximum buffer size needed is returned

Example

```
WOLFSSL ssl;
unsigned char* buffer;
size_t bufferSz;
size_t ret;
bufferSz  = wolfSSL_get_client_random(ssl, NULL, 0);
buffer = malloc(bufferSz);
ret  = wolfSSL_get_client_random(ssl, buffer, bufferSz);
// check ret value
```

C.58.4.436 function wolfSSL_CTX_get_default_passwd_cb

```
wc_pem_password_cb * wolfSSL_CTX_get_default_passwd_cb(
    WOLFSSL_CTX * ctx
)
```

This is a getter function for the password callback set in ctx.

Parameters:

- **ctx** WOLFSSL_CTX structure to get call back from.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free

Return:

- func On success returns the callback function.
- NULL If ctx is NULL then NULL is returned.

Example

```
WOLFSSL_CTX* ctx;
wc_pem_password_cb cb;
// setup ctx
cb = wolfSSL_CTX_get_default_passwd_cb(ctx);
//use cb
```

C.58.4.437 function wolfSSL_CTX_get_default_passwd_cb_userdata

```
void * wolfSSL_CTX_get_default_passwd_cb_userdata(
    WOLFSSL_CTX * ctx
)
```

This is a getter function for the password callback user data set in ctx.

Parameters:

- **ctx** WOLFSSL_CTX structure to get user data from.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free

Return:

- pointer On success returns the user data pointer.
- NULL If ctx is NULL then NULL is returned.

Example

```
WOLFSSL_CTX* ctx;
void* data;
// setup ctx
data = wolfSSL_CTX_get_default_passwd_cb(ctx);
//use data
```

C.58.4.438 function wolfSSL_PEM_read_bio_X509_AUX

```
WOLFSSL_X509 * wolfSSL_PEM_read_bio_X509_AUX(
    WOLFSSL_BIO * bp,
    WOLFSSL_X509 ** x,
    wc_pem_password_cb * cb,
    void * u
)
```

This function behaves the same as wolfSSL_PEM_read_bio_X509. AUX signifies containing extra information such as trusted/rejected use cases and friendly name for human readability.

Parameters:

- **bp** WOLFSSL_BIO structure to get PEM buffer from.

- **x** if setting WOLFSSL_X509 by function side effect.
- **cb** password callback.
- **u** NULL terminated user password.

See: wolfSSL_PEM_read_bio_X509

Return:

- WOLFSSL_X509 on successfully parsing the PEM buffer a WOLFSSL_X509 structure is returned.
- Null if failed to parse PEM buffer.

Example

```
WOLFSSL_BIO* bio;
WOLFSSL_X509* x509;
// setup bio
X509 = wolfSSL_PEM_read_bio_X509_AUX(bio, NULL, NULL, NULL);
//check x509 is not null and then use it
```

C.58.4.439 function wolfSSL_CTX_set_tmp_dh

```
long wolfSSL_CTX_set_tmp_dh(
    WOLFSSL_CTX * ctx,
    WOLFSSL_DH * dh
)
```

Initializes the WOLFSSL_CTX structure's dh member with the Diffie-Hellman parameters.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **dh** a pointer to a WOLFSSL_DH structure.

See: wolfSSL_BN_bn2bin

Return:

- SSL_SUCCESS returned if the function executed successfully.
- BAD_FUNC_ARG returned if the ctx or dh structures are NULL.
- SSL_FATAL_ERROR returned if there was an error setting a structure value.
- MEMORY_E returned if their was a failure to allocate memory.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL_DH* dh;
…
return wolfSSL_CTX_set_tmp_dh(ctx, dh);
```

C.58.4.440 function wolfSSL_PEM_read_bio_DSAparams

```
WOLFSSL_DSA * wolfSSL_PEM_read_bio_DSAparams(
    WOLFSSL_BIO * bp,
    WOLFSSL_DSA ** x,
    wc_pem_password_cb * cb,
    void * u
)
```

This function get the DSA parameters from a PEM buffer in bio.

Parameters:

- **bio** pointer to the WOLFSSL_BIO structure for getting PEM memory pointer.
- **x** pointer to be set to new WOLFSSL_DSA structure.
- **cb** password callback function.
- **u** null terminated password string.

See: none

Return:

- WOLFSSL_DSA on successfully parsing the PEM buffer a WOLFSSL_DSA structure is created and returned.
- Null if failed to parse PEM buffer.

Example

```
WOLFSSL_BIO* bio;
WOLFSSL_DSA* dsa;
// setup bio
dsa = wolfSSL_PEM_read_bio_DSAparams(bio, NULL, NULL, NULL);

// check dsa is not NULL and then use dsa
```

C.58.4.441 function wolfSSL_ERR_peek_last_error

```
unsigned long wolfSSL_ERR_peek_last_error(
    void
)
```

This function returns the absolute value of the last error from WOLFSSL_ERROR encountered.

Parameters:

- **none** No parameters.

See: wolfSSL_ERR_print_errors_fp

Return: error Returns absolute value of last error.

Example

```
unsigned long err;
...
err = wolfSSL_ERR_peek_last_error();
// inspect err value
```

C.58.4.442 function WOLF_STACK_OF

```
WOLF_STACK_OF(
    WOLFSSL_X509
) const
```

This function gets the peer's certificate chain.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_subject_name
- wolfSSL_X509_get_isCA

Return:

- pointer returns a pointer to the peer's Certificate stack.
- NULL returned if no peer certificate.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
wolfSSL_connect(ssl);
STACK_OF(WOLFSSL_X509)* chain = wolfSSL_get_peer_cert_chain(ssl);
ifchain){
    // You have a pointer to the peer certificate chain
}
```

C.58.4.443 function wolfSSL_CTX_clear_options

```
long wolfSSL_CTX_clear_options(
    WOLFSSL_CTX * ctx,
    long opt
)
```

This function resets option bits of WOLFSSL_CTX object.

Parameters:

- **ctx** pointer to the SSL context.

See:

- wolfSSL_CTX_new
- wolfSSL_new
- wolfSSL_free

Return: option new option bits

Example

```
WOLFSSL_CTX* ctx = 0;
...
wolfSSL_CTX_clear_options(ctx, SSL_OP_NO_TLSv1);
```

C.58.4.444 function wolfSSL_set_jobject

```
int wolfSSL_set_jobject(
    WOLFSSL * ssl,
    void * objPtr
)
```

This function sets the jObjectRef member of the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **objPtr** a void pointer that will be set to jObjectRef.

See: wolfSSL_get_jobject

Return:

- SSL_SUCCESS returned if jObjectRef is properly set to objPtr.
- SSL_FAILURE returned if the function did not properly execute and jObjectRef is not set.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = WOLFSSL_new();
void* objPtr = &obj;
...
if(wolfSSL_set_jobject(ssl, objPtr)){
    // The success case
}
```

C.58.4.445 function wolfSSL_get_jobject

```
void * wolfSSL_get_jobject(
    WOLFSSL * ssl
)
```

This function returns the jObjectRef member of the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_set_jobject

Return:

- value If the WOLFSSL struct is not NULL, the function returns the jObjectRef value.
- NULL returned if the WOLFSSL struct is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL(ctx);
...
void* jobject = wolfSSL_get_jobject(ssl);

if(jobject != NULL){
    // Success case
}
```

C.58.4.446 function wolfSSL_set_msg_callback

```
int wolfSSL_set_msg_callback(
    WOLFSSL * ssl,
    SSL_Msg_Cb cb
)
```

This function sets a callback in the ssl. The callback is to observe handshake messages. NULL value of cb resets the callback.

Parameters:

- **ssl** WOLFSSL structure to set callback argument.

See: wolfSSL_set_msg_callback_arg

Return:

- SSL_SUCCESS On success.
- SSL_FAILURE If an NULL ssl passed in.

Example

```
static cb(int write_p, int version, int content_type,
const void *buf, size_t len, WOLFSSL *ssl, void *arg)
…
WOLFSSL* ssl;
ret  = wolfSSL_set_msg_callback(ssl, cb);
// check ret
```

C.58.4.447 function wolfSSL_set_msg_callback_arg

```
int wolfSSL_set_msg_callback_arg(
    WOLFSSL * ssl,
    void * arg
)
```

This function sets associated callback context value in the ssl. The value is handed over to the callback argument.

Parameters:

- **ssl** WOLFSSL structure to set callback argument.

See: wolfSSL_set_msg_callback

Return: none No return.

Example

```
static cb(int write_p, int version, int content_type,
const void *buf, size_t len, WOLFSSL *ssl, void *arg)
…
WOLFSSL* ssl;
ret  = wolfSSL_set_msg_callback(ssl, cb);
// check ret
wolfSSL_set_msg_callback(ssl, arg);
```

C.58.4.448 function wolfSSL_X509_get_next_altname

```
char * wolfSSL_X509_get_next_altname(
    WOLFSSL_X509 *
)
```

This function returns the next, if any, altname from the peer certificate.

Parameters:

- **cert** a pointer to the wolfSSL_X509 structure.

See:

- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_subject_name
- wolfSSL_X509_get_ext_d2i

Return:

- NULL if there is not a next altname.
- cert->altNamesNext->name from the WOLFSSL_X509 structure that is a string value from the altName list is returned if it exists.

Warning: The returned value is a bare C string with no length. A dNSName, rfc822Name, or uniformResourceIdentifier SAN may legally contain an embedded NUL byte (RFC 6125 Sec. 6.3 treats such a name as an invalid presented identifier, not a malformed certificate, so the certificate still parses),

which silently truncates the returned string. Do not use strlen or strcmp on the result for security comparisons; use wolfSSL_X509_get_ext_d2i and the GENERAL_NAME ASN1_STRING length instead.

Example

```
WOLFSSL_X509 x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                                                    DYNAMIC_TYPE_X509);
…
int x509NextAltName = wolfSSL_X509_get_next_altname(x509);
if(x509NextAltName == NULL){
        //There isn't another alt name
}
```

C.58.4.449 function wolfSSL_X509_get_notBefore

```
WOLFSSL_ASN1_TIME * wolfSSL_X509_get_notBefore(
    WOLFSSL_X509 *
)
```

The function checks to see if x509 is NULL and if it's not, it returns the notBefore member of the x509 struct.

Parameters:

- **x509** a pointer to the WOLFSSL_X509 struct.

See: wolfSSL_X509_get_notAfter

Return:

- pointer to struct with ASN1_TIME to the notBefore member of the x509 struct.
- NULL the function returns NULL if the x509 structure is NULL.

Example

```
WOLFSSL_X509* x509 = (WOLFSSL_X509)XMALLOC(sizeof(WOLFSSL_X509), NULL,
DYNAMIC_TYPE_X509) ;
…
const WOLFSSL_ASN1_TIME* notAfter = wolfSSL_X509_get_notBefore(x509);
if(notAfter == NULL){
        //The x509 object was NULL
}
```

C.58.4.450 function wolfSSL_connect

```
int wolfSSL_connect(
    WOLFSSL * ssl
)
```

This function is called on the client side and initiates an SSL/TLS handshake with a server. When this function is called, the underlying communication channel has already been set up. wolfSSL_connect() works with both blocking and non_blocking I/O. When the underlying I/O is non_blocking, wolfSSL_connect() will return when the underlying I/O could not satisfy the needs of wolfSSL_connect to continue the handshake. In this case, a call to wolfSSL_get_error() will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_connect() when the underlying I/O is ready and wolfSSL will pick up where it left off. When using a non_blocking socket, nothing needs to be done, but select() can be used to check for the required condition. If the underlying I/O is blocking, wolfSSL_connect() will only return once the handshake has been finished or an error occurred. wolfSSL takes a different approach to certificate verification than OpenSSL does. The default policy for the client is to verify the server, this means

that if you don't load CAs to verify the server you'll get a connect error, unable to verify (_155). It you want to mimic OpenSSL behavior of having SSL_connect succeed even if verifying the server fails and reducing security you can do this by calling: SSL_CTX_set_verify(ctx, SSL_VERIFY_NONE, 0); before calling SSL_new(); Though it's not recommended.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_get_error
- wolfSSL_accept

Return:

- SSL_SUCCESS If successful.
- SSL_FATAL_ERROR will be returned if an error occurred. To get a more detailed error code, call wolfSSL_get_error().
- BAD_FUNC_ARG will be returned when ssl is NULL.

If decrypted application data is waiting to be read, for example data received during a secure renegotiation, the handshake is not continued and wolfSSL_get_error() returns APP_DATA_READY. Read the data with wolfSSL_read() and then call wolfSSL_connect() again.

Example

```
int ret = 0;
int err = 0;
WOLFSSL* ssl;
char buffer[80];
...
ret = wolfSSL_connect(ssl);
if (ret != SSL_SUCCESS) {
err = wolfSSL_get_error(ssl, ret);
printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
}
```

C.58.4.451 function wolfSSL_send_hrr_cookie

```
int wolfSSL_send_hrr_cookie(
    WOLFSSL * ssl,
    const unsigned char * secret,
    unsigned int secretSz
)
```

This function is called on the server side to indicate that a HelloRetryRequest message must contain a Cookie and, in case of using protocol DTLS v1.3, that the handshake will always include a cookie exchange. Please note that when using protocol DTLS v1.3, the cookie exchange is enabled by default. The Cookie holds a hash of the current transcript so that another server process can handle the ClientHello in reply. The secret is used when generating the integrity check on the Cookie data. This replaces or regenerates the HRR secret, then delegates to wolfSSL_enable_cookie(). Changing cookie mode after handshake processing starts is unsupported; rotating a secret while cookies are already enabled remains supported. DTLS 1.2_only callers should use wolfSSL_enable_cookie() and wolfSSL_DTLS_SetCookieSecret().

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

- **secret** a pointer to a buffer holding the secret. Passing NULL indicates to generate a new random secret.
- **secretSz** Size of the secret in bytes. Passing 0 indicates to use the default size: the digest size of the hash the cookie MAC uses, WC_SHA256_DIGEST_SIZE, or when SHA-256 is not available WC_SHA384_DIGEST_SIZE, WC_SHA512_DIGEST_SIZE or WC_SM3_DIGEST_SIZE, in that order.

See:

- wolfSSL_new
- wolfSSL_disable_hrr_cookie

Return:

- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3.
- SIDE_ERROR if called with a client.
- BAD_STATE_E if the DTLS handshake already committed to stateful processing, as for wolfSSL_enable_cookie(), or if a secret must be generated (secret is NULL, or a DTLS 1.3 server is missing its DTLS 1.2 secret) and the session's RNG has been released. The installed secret is left unchanged.
- WOLFSSL_SUCCESS if successful.
- MEMORY_ERROR if allocating dynamic memory for storing secret failed.
- Another -ve value on internal error. Every failure is all or nothing: the secret in use and the cookie policy are left exactly as they were found, so the object stays usable.

Example

```
int ret;
WOLFSSL* ssl;
char secret[32];
...
ret = wolfSSL__send_hrr_cookie(ssl, secret, sizeof(secret));
if (ret != WOLFSSL_SUCCESS) {
    // failed to set use of Cookie and secret
}
```

C.58.4.452 function wolfSSL_set_hrr_cookie_secret_secondary

```
int wolfSSL_set_hrr_cookie_secret_secondary(
    WOLFSSL * ssl,
    const unsigned char * secret,
    unsigned int secretSz
)
```

This function sets a secondary HelloRetryRequest cookie secret on a DTLS 1.3 server. It is used only when verifying a received cookie, and only if the primary secret (set with wolfSSL_send_hrr_cookie()) fails to verify it. This lets an application rotate the cookie secret on a stateless DTLS 1.3 server without rejecting clients whose cookie was issued under the previous secret: install the new secret as the primary and the previous secret here for an overlap window. The secondary secret is never used to issue cookies. This API is DTLS only; TLS 1.3 over a reliable transport does not verify cookies statelessly across the HelloRetryRequest exchange.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **secret** a pointer to a buffer holding the secondary secret. Passing NULL (or a secretSz of 0) clears any previously set secondary secret.
- **secretSz** Size of the secret in bytes.

See:

- wolfSSL_send_hrr_cookie
- wolfSSL_disable_hrr_cookie

Return:

- BAD_FUNC_ARG if ssl is NULL, not using TLS v1.3, or not using DTLS.
- SIDE_ERROR if called with a client.
- WOLFSSL_SUCCESS if successful.
- MEMORY_ERROR if allocating dynamic memory for storing secret failed. The secondary secret already in use, if any, is kept.

Example

```
int ret;
WOLFSSL* ssl;
char newSecret[32];
char oldSecret[32];
...
// rotate: new secret becomes primary, previous secret stays valid
wolfSSL_send_hrr_cookie(ssl, newSecret, sizeof(newSecret));
ret = wolfSSL_set_hrr_cookie_secret_secondary(ssl, oldSecret,
    sizeof(oldSecret));
if (ret != WOLFSSL_SUCCESS) {
    // failed to set the secondary cookie secret
}
```

C.58.4.453 function wolfSSL_disable_hrr_cookie

```
int wolfSSL_disable_hrr_cookie(
    WOLFSSL * ssl
)
```

This function is called on the server side to indicate that a HelloRetryRequest message must NOT contain a Cookie and that, if using protocol DTLS v1.3, a cookie exchange will not be included in the handshake. Please note that not doing a cookie exchange when using protocol DTLS v1.3 can make the server susceptible to DoS/Amplification attacks. This delegates to wolfSSL_disable_cookie(): on a DTLS object it also disables the DTLS 1.2 HelloVerifyRequest exchange, and it fails with BAD_STATE_E once the handshake has committed to stateful processing.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_disable_cookie
- wolfSSL_send_hrr_cookie

Return:

- WOLFSSL_SUCCESS if successful
- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3
- SIDE_ERROR if invoked on a client
- BAD_STATE_E if the DTLS handshake already committed to stateful processing

C.58.4.454 function wolfSSL_CTX_no_ticket_TLSv13

```
int wolfSSL_CTX_no_ticket_TLSv13(
    WOLFSSL_CTX * ctx
)
```

This function is called on the server to stop it from sending a resumption session ticket once the handshake is complete.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().

See: wolfSSL_no_ticket_TLSv13

Return:

- BAD_FUNC_ARG if ctx is NULL or not using TLS v1.3.
- SIDE_ERROR if called with a client.
- 0 if successful.

Example

```
int ret;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_no_ticket_TLSv13(ctx);
if (ret != 0) {
    // failed to set no ticket
}
```

C.58.4.455 function wolfSSL_no_ticket_TLSv13

```
int wolfSSL_no_ticket_TLSv13(
    WOLFSSL * ssl
)
```

This function is called on the server to stop it from sending a resumption session ticket once the handshake is complete.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_CTX_no_ticket_TLSv13

Return:

- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3.
- SIDE_ERROR if called with a client.
- 0 if successful.

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_no_ticket_TLSv13(ssl);
if (ret != 0) {
    // failed to set no ticket
}
```

C.58.4.456 function wolfSSL_CTX_no_dhe_psk

```
int wolfSSL_CTX_no_dhe_psk(
    WOLFSSL_CTX * ctx
)
```

This function is called on a TLS v1.3 wolfSSL context to disallow Diffie-Hellman (DH) style key exchanges when handshakes are using pre-shared keys for authentication.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().

See: wolfSSL_no_dhe_psk

Return:

- BAD_FUNC_ARG if ctx is NULL or not using TLS v1.3.
- 0 if successful.

Example

```
int ret;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_no_dhe_psk(ctx);
if (ret != 0) {
    // failed to set no DHE for PSK handshakes
}
```

C.58.4.457 function wolfSSL_no_dhe_psk

```
int wolfSSL_no_dhe_psk(
    WOLFSSL * ssl
)
```

This function is called on a TLS v1.3 client or server wolfSSL to disallow Diffie-Hellman (DH) style key exchanges when handshakes are using pre-shared keys for authentication.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_CTX_no_dhe_psk

Return:

- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3.
- 0 if successful.

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_no_dhe_psk(ssl);
if (ret != 0) {
    // failed to set no DHE for PSK handshakes
}
```

C.58.4.458 function wolfSSL_CTX_require_psk

```
int wolfSSL_CTX_require_psk(
    WOLFSSL_CTX * ctx
)
```

This function is called on a TLS v1.3 / DTLS v1.3 context to require that an external Pre-Shared Key is negotiated for the handshake to succeed. When set, a handshake that completes without negotiating

an external PSK is aborted with PSK_MISSING_ERROR instead of falling back to a certificate handshake, so the PSK acts as an additional security factor. The requirement keys off the external-PSK callback (it has no effect unless one is registered) and session-ticket resumption is exempt. To preserve forward secrecy a mandatory external PSK must also use an (EC)DHE key exchange; a pure psk_ke handshake is rejected with PSK_KEY_ERROR. This applies to TLS 1.3 and DTLS 1.3 only; in (D)TLS 1.2 the use of a PSK is determined by the negotiated cipher suite, so a mandatory PSK is instead configured by restricting the cipher suite list to (preferably (EC)DHE-)PSK suites.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

See: wolfSSL_require_psk

Return:

- BAD_FUNC_ARG if ctx is NULL or not at least TLS v1.3.
- 0 if successful.

Note: In builds compiled without external-PSK and without session-ticket support (NO_PSK defined and HAVE_SESSION_TICKET undefined) the requirement cannot be enforced; the function still returns 0 but has no effect.

Warning: Because the requirement can only be enforced for (D)TLS 1.3, this function also disables version downgrade on the context so it cannot silently fall back to (D)TLS 1.2 and complete a handshake without a PSK. A peer that does not support (D)TLS 1.3 will therefore fail to connect.

Example

```
int ret;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_require_psk(ctx);
if (ret != 0) {
    // failed to make a PSK mandatory
}
```

C.58.4.459 function wolfSSL_require_psk

```
int wolfSSL_require_psk(
    WOLFSSL * ssl
)
```

This function is called on a TLS v1.3 / DTLS v1.3 wolfSSL object to require that an external Pre-Shared Key is negotiated for the handshake to succeed. See wolfSSL_CTX_require_psk() for the full behaviour.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_CTX_require_psk

Return:

- BAD_FUNC_ARG if ssl is NULL or not at least TLS v1.3.
- 0 if successful.

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_require_psk(ssl);
```

```
if (ret != 0) {
    // failed to make a PSK mandatory
}
```

C.58.4.460 function wolfSSL_update_keys

```
int wolfSSL_update_keys(
    WOLFSSL * ssl
)
```

This function is called on a TLS v1.3 client or server wolfSSL to force the rollover of keys. A KeyUpdate message is sent to the peer and new keys are calculated for encryption. The peer will send back a KeyUpdate message and the new decryption keys will then be calculated. This function can only be called after a handshake has been completed.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_write

Return:

- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3.
- WANT_WRITE if the writing is not ready.
- BAD_STATE_E if the connection has already performed the maximum number of key updates. RFC 9846, Section 4.7.3 caps a TLS 1.3 sender at 2^48-1 key updates; beyond that the connection must be closed rather than rekeyed. Note that a KeyUpdate arriving from the peer with request_update set is ignored once this cap is reached, rather than failing the connection, so only an application-initiated update reports this error.
- WOLFSSL_SUCCESS if successful.

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_update_keys(ssl);
if (ret == WANT_WRITE) {
    // need to call again when I/O ready
}
else if (ret != WOLFSSL_SUCCESS) {
    // failed to send key update
}
```

C.58.4.461 function wolfSSL_key_update_response

```
int wolfSSL_key_update_response(
    WOLFSSL * ssl,
    int * required
)
```

This function is called on a TLS v1.3 client or server wolfSSL to determine whether a rollover of keys is in progress. When wolfSSL_update_keys() is called, a KeyUpdate message is sent and the encryption key is updated. The decryption key is updated when the response is received.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

- **required** 0 when no key update response is required. 1 when a key update response from the peer is still outstanding.

See: wolfSSL_update_keys

Return:

- 0 on successful.
- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3.

Example

```
int ret;
WOLFSSL* ssl;
int required;
...
ret = wolfSSL_key_update_response(ssl, &required);
if (ret != 0) {
    // bad parameters
}
if (required) {
    // encrypt Key updated, awaiting response to change decrypt key
}
```

C.58.4.462 function wolfSSL_CTX_allow_post_handshake_auth

```
int wolfSSL_CTX_allow_post_handshake_auth(
    WOLFSSL_CTX * ctx
)
```

This function is called on a TLS v1.3 client wolfSSL context to allow a client certificate to be sent post handshake upon request from server. This is useful when connecting to a web server that has some pages that require client authentication and others that don't.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().

See:

- wolfSSL_allow_post_handshake_auth
- wolfSSL_request_certificate

Return:

- BAD_FUNC_ARG if ctx is NULL or not using TLS v1.3.
- SIDE_ERROR if called with a server.
- 0 if successful.

Example

```
int ret;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_allow_post_handshake_auth(ctx);
if (ret != 0) {
    // failed to allow post handshake authentication
}
```

C.58.4.463 function wolfSSL_allow_post_handshake_auth

```
int wolfSSL_allow_post_handshake_auth(
    WOLFSSL * ssl
)
```

This function is called on a TLS v1.3 client wolfSSL to allow a client certificate to be sent post handshake upon request from server. A Post-Handshake Client Authentication extension is sent in the ClientHello. This is useful when connecting to a web server that has some pages that require client authentication and others that don't.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_CTX_allow_post_handshake_auth
- wolfSSL_request_certificate

Return:

- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3.
- SIDE_ERROR if called with a server.
- BAD_STATE_E if called after the handshake has started.
- 0 if successful.

This function must be called before wolfSSL_connect() on the WOLFSSL object.

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_allow_post_handshake_auth(ssl);
if (ret != 0) {
    // failed to allow post handshake authentication
}
```

C.58.4.464 function wolfSSL_request_certificate

```
int wolfSSL_request_certificate(
    WOLFSSL * ssl
)
```

This function requests a client certificate from the TLS v1.3 client. This is useful when a web server is serving some pages that require client authentication and others that don't. A maximum of 256 requests can be sent on a connection.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_allow_post_handshake_auth
- wolfSSL_write

Return:

- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3.
- WANT_WRITE if the writing is not ready.
- SIDE_ERROR if called with a client.

- NOT_READY_ERROR if called when the handshake is not finished.
- POST_HAND_AUTH_ERROR if posthandshake authentication is disallowed.
- MEMORY_E if dynamic memory allocation fails.
- WOLFSSL_SUCCESS if successful.

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_request_certificate(ssl);
if (ret == WANT_WRITE) {
    // need to call again when I/O ready
}
else if (ret != WOLFSSL_SUCCESS) {
    // failed to request a client certificate
}
```

C.58.4.465 function wolfSSL_CTX_set1_sigalgs_list

```
int wolfSSL_CTX_set1_sigalgs_list(
    WOLFSSL_CTX * ctx,
    const char * list
)
```

This function sets the list of signature algorithms to allow on a wolfSSL context in order of preference. The list is a null-terminated text string and a colon-delimited list, where each element is of the form "+" (for example "RSA-PSS+SHA256:ECDSA+SHA384"). The Edwards-curve algorithms "ED25519" and "ED448" are written without a digest suffix, since the hash is implied by the algorithm. The previous list stored in the context is replaced.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().
- **list** a colon-delimited list of "+" elements (or "ED25519" / "ED448" without a digest).

See:

- wolfSSL_set1_sigalgs_list
- wolfSSL_CTX_set1_groups_list

Return:

- WOLFSSL_SUCCESS if successful.
- WOLFSSL_FAILURE if a pointer parameter is NULL, allocation of the suites structure fails, or any token in the list is not recognized or not supported by the current build.

Recognized public-key tokens and digest tokens, together with the build options that must be enabled for each token to be accepted, are listed in the tables below.

Public-key tokens:

Token	Required build option
RSA	!NO_RSA
RSA-PSS / PSS	!NO_RSA and WC_RSA_PSS
ECDSA	HAVE_ECC
ED25519	HAVE_ED25519 (no digest suffix)
ED448	HAVE_ED448 (no digest suffix)
DSA	!NO_DSA

Token	Required build option
SM2	WOLFSSL_SM2 and WOLFSSL_SM3 (digest is SM3)

Digest tokens:

Token	Required build option
SHA256	!NO_SHA256
SHA384	WOLFSSL_SHA384
SHA512	WOLFSSL_SHA512
SHA224	WOLFSSL_SHA224
SM3	WOLFSSL_SM3
SHA1	!NO_SHA, plus !NO_OLD_TLS or WOLFSSL_ALLOW_TLS_SHA1

Notes for TLS 1.3: per RFC 8446, RSA PKCS#1 v1.5, DSA, SHA-1, and SHA-224 cannot be used as handshake signatures and will be filtered out at negotiation time even if listed. Specifying "RSA-PSS+SHAxxx" causes both the rsa_pss_rsae_shaxxx and rsa_pss_pss_shaxxx schemes to be added. Brainpool ECDSA signature schemes (RFC 8734) cannot be selected through this string interface; they are negotiated automatically when HAVE_ECC_BRAINPOOL is enabled.

Example

```
int ret;
WOLFSSL_CTX* ctx;
const char* list = "RSA-PSS+SHA256:ECDSA+SHA384:ED25519";
...
ret = wolfSSL_CTX_set1_sigalgs_list(ctx, list);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set signature algorithm list
}
```

C.58.4.466 function wolfSSL_set1_sigalgs_list

```
int wolfSSL_set1_sigalgs_list(
    WOLFSSL * ssl,
    const char * list
)
```

This function sets the list of signature algorithms to allow on a wolfSSL session in order of preference. The list format and the set of recognized public-key and digest tokens are identical to those documented for wolfSSL_CTX_set1_sigalgs_list(); refer to that function for the full token tables and TLS 1.3 caveats. The previous list stored in the session is replaced.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **list** a colon-delimited list of "+" elements (or "ED25519" / "ED448" without a digest).

See:

- wolfSSL_CTX_set1_sigalgs_list
- wolfSSL_set1_groups_list

Return:

- WOLFSSL_SUCCESS if successful.

- WOLFSSL_FAILURE if a pointer parameter is NULL, allocation of the suites structure fails, or any token in the list is not recognized or not supported by the current build.

Example

```
int ret;
WOLFSSL* ssl;
const char* list = "RSA-PSS+SHA256:ECDSA+SHA384:ED25519";
...
ret = wolfSSL_set1_sigalgs_list(ssl, list);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set signature algorithm list
}
```

C.58.4.467 function wolfSSL_CTX_set1_groups_list

```
int wolfSSL_CTX_set1_groups_list(
    WOLFSSL_CTX * ctx,
    const char * list
)
```

This function sets the list of key-exchange groups (named elliptic curves and KEMs) to allow on a wolfSSL context in order of preference. The list is a null-terminated, colon-delimited text string of group names, for example "P-384:P-256:X25519". Call this function to set the key-exchange parameters used with TLS v1.3 connections (the function is compiled in only when HAVE_ECC, WOLFSSL_TLS13, and HAVE_SUPPORTED_CURVES are defined).

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().
- **list** a string that is a colon-delimited list of key-exchange group names.

See:

- wolfSSL_set1_groups_list
- wolfSSL_CTX_set_groups
- wolfSSL_set_groups
- wolfSSL_UseKeyShare
- wolfSSL_preferred_group

Return:

- WOLFSSL_FAILURE if pointer parameters are NULL, there are more than WOLFSSL_MAX_GROUP_COUNT groups, a group name is not recognized or not using TLS v1.3.
- WOLFSSL_SUCCESS if successful.

Recognized group names and the build options each one requires are listed below. Names are matched case-sensitively against the table.

NIST / SEC curves (require HAVE_ECC):

Name	Curve / Group
P-160	secp160r1
P-160-2	secp160r2
P-192	secp192r1 (prime192v1)
P-224	secp224r1
P-256	secp256r1 (prime256v1) — also accepted as "prime256v1" / "secp256r1"
P-384	secp384r1 — also accepted as "secp384r1"

Name	Curve / Group
P-521	secp521r1 — also accepted as “secp521r1”
K-160	secp160k1
K-192	secp192k1
K-224	secp224k1
K-256	secp256k1

Brainpool curves (require HAVE_ECC plus WOLFSSL_CUSTOM_CURVES and HAVE_ECC_BRAINPOOL — typically enabled by –enable-ecccustcurves=all):

Name	Curve
B-256	brainpoolP256r1
B-384	brainpoolP384r1
B-512	brainpoolP512r1

Edwards / Montgomery curves:

Name	Required build option
X25519	HAVE_CURVE25519
X448	HAVE_CURVE448

SM2 (requires WOLFSSL_SM2):

Name	Group
SM2	sm2p256v1
sm2p256v1	sm2p256v1 (alias)

ML-KEM (post-quantum) groups (require WOLFSSL_HAVE_MLKEM and !WOLFSSL_NO_ML_KEM):

Name
ML_KEM_512
ML_KEM_768
ML_KEM_1024

ML-KEM hybrid groups additionally require HAVE_ECC together with either WOLFSSL_WC_MLKEM, and WOLFSSL_PQC_HYBRIDS (or WOLFSSL_EXTRA_PQC_HYBRIDS for the “extra” set):

Name	Hybrid flag set
SecP256r1MLKEM768	WOLFSSL_PQC_HYBRIDS
SecP384r1MLKEM1024	WOLFSSL_PQC_HYBRIDS
X25519MLKEM768	WOLFSSL_PQC_HYBRIDS
SecP256r1MLKEM512	WOLFSSL_EXTRA_PQC_HYBRIDS
SecP384r1MLKEM768	WOLFSSL_EXTRA_PQC_HYBRIDS
SecP521r1MLKEM1024	WOLFSSL_EXTRA_PQC_HYBRIDS

COPYRIGHT ©2024 wolfSSL Inc.

Name	Hybrid flag set
X25519MLKEM512	WOLFSSL_EXTRA_PQC_HYBRIDS
X448MLKEM768	WOLFSSL_EXTRA_PQC_HYBRIDS

Legacy Kyber groups (require WOLFSSL_MLKEM_KYBER; hybrids additionally require HAVE_ECC together with WOLFSSL_WC_MLKEM):

Name
KYBER_LEVEL1
KYBER_LEVEL3
KYBER_LEVEL5
P256_KYBER_LEVEL1
P256_KYBER_LEVEL3
P384_KYBER_LEVEL3
P521_KYBER_LEVEL5
X25519_KYBER_LEVEL1
X25519_KYBER_LEVEL3
X448_KYBER_LEVEL3

In addition to the names above, when HAVE_FIPS and HAVE_SELFTEST are not defined, any curve name registered with wolfCrypt (looked up via wc_ecc_get_curve_idx_from_name(), e.g. “brainpoolP256r1”) is also accepted.

The order of the names in the list is preserved and used as the local preference order for KeyShare selection in TLS 1.3.

Example

```
int ret;
WOLFSSL_CTX* ctx;
const char* list = "P-384:P-256";
...
ret = wolfSSL_CTX_set1_groups_list(ctx, list);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set group list
}
```

C.58.4.468 function wolfSSL_set1_groups_list

```
int wolfSSL_set1_groups_list(
    WOLFSSL * ssl,
    const char * list
)
```

This function sets the list of key-exchange groups (named elliptic curves and KEMs) to allow on a wolfSSL session in order of preference. The list format and the set of recognized group names are identical to those documented for wolfSSL_CTX_set1_groups_list(); refer to that function for the full token tables and required build options.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **list** a string that is a colon separated list of key exchange groups.

See:

- wolfSSL_CTX_set1_groups_list
- wolfSSL_CTX_set_groups
- wolfSSL_set_groups
- wolfSSL_UseKeyShare
- wolfSSL_preferred_group

Return:

- WOLFSSL_FAILURE if pointer parameters are NULL, there are more than WOLFSSL_MAX_GROUP_COUNT groups, a group name is not recognized or not using TLS v1.3.
- WOLFSSL_SUCCESS if successful.

Example

```
int ret;
WOLFSSL* ssl;
const char* list = "P-384:P-256";
...
ret = wolfSSL_CTX_set1_groups_list(ssl, list);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set group list
}
```

C.58.4.469 function wolfSSL_preferred_group

```
int wolfSSL_preferred_group(
    WOLFSSL * ssl
)
```

This function returns the key exchange group the client prefers to use in the TLS v1.3 handshake. Call this function to after a handshake is complete to determine which group the server prefers so that this information can be used in future connections to pre-generate a key pair for key exchange.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_UseKeyShare
- wolfSSL_CTX_set_groups
- wolfSSL_set_groups
- wolfSSL_CTX_set1_groups_list
- wolfSSL_set1_groups_list

Return:

- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3.
- SIDE_ERROR if called with a server.
- NOT_READY_ERROR if called before handshake is complete.
- Group identifier if successful.

Example

```
int ret;
int group;
WOLFSSL* ssl;
...
ret = wolfSSL_CTX_set1_groups_list(ssl)
```

```
if (ret < 0) {
    // failed to get group
}
group = ret;
```

C.58.4.470 function wolfSSL_CTX_set_groups

```
int wolfSSL_CTX_set_groups(
    WOLFSSL_CTX * ctx,
    int * groups,
    int count
)
```

This function sets the list of key-exchange groups (named elliptic curves and KEMs) to allow on a wolfSSL context in order of preference. The list is an array of named-group identifiers (see the table below) and `count` is the number of identifiers in the array. Use this function to set the key-exchange parameters used by TLS v1.3 connections; the order of the array becomes the local KeyShare preference order.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().
- **groups** a list of key-exchange groups by identifier.
- **count** the number of identifiers in `groups` (must not exceed WOLFSSL_MAX_GROUP_COUNT).

See:

- wolfSSL_set_groups
- wolfSSL_UseKeyShare
- wolfSSL_CTX_set1_groups_list
- wolfSSL_set1_groups_list
- wolfSSL_preferred_group

Return:

- BAD_FUNC_ARG if a pointer parameter is NULL, `count` exceeds WOLFSSL_MAX_GROUP_COUNT, or the underlying method is not a TLS method.
- WOLFSSL_SUCCESS if successful.

Recognized identifiers and the build options each one requires are listed below. The identifiers are defined in the anonymous enum in <wolfssl/ssl.h>.

NIST / SEC curves (require HAVE_ECC):

Identifier	Curve / Group
WOLFSSL_ECC_SECP160K1	secp160k1
WOLFSSL_ECC_SECP160R1	secp160r1
WOLFSSL_ECC_SECP160R2	secp160r2
WOLFSSL_ECC_SECP192K1	secp192k1
WOLFSSL_ECC_SECP192R1	secp192r1 (prime192v1)
WOLFSSL_ECC_SECP224K1	secp224k1
WOLFSSL_ECC_SECP224R1	secp224r1
WOLFSSL_ECC_SECP256K1	secp256k1
WOLFSSL_ECC_SECP256R1	secp256r1 (prime256v1)
WOLFSSL_ECC_SECP384R1	secp384r1
WOLFSSL_ECC_SECP521R1	secp521r1

Brainpool curves (require HAVE_ECC plus WOLFSSL_CUSTOM_CURVES and HAVE_ECC_BRAINPOOL — typically enabled by –enable-ecccustcurves=all):

Identifier	Curve	Notes
WOLFSSL_ECC_BRAINPOOLP256R1	brainpoolP256r1	TLS 1.2 group ID 26
WOLFSSL_ECC_BRAINPOOLP384R1	brainpoolP384r1	TLS 1.2 group ID 27
WOLFSSL_ECC_BRAINPOOLP512R1	brainpoolP512r1	TLS 1.2 group ID 28
WOLFSSL_ECC_BRAINPOOLP256R1TLS13	brainpoolP256r1	RFC 8734 TLS 1.3 ID
WOLFSSL_ECC_BRAINPOOLP384R1TLS13	brainpoolP384r1	RFC 8734 TLS 1.3 ID
WOLFSSL_ECC_BRAINPOOLP512R1TLS13	brainpoolP512r1	RFC 8734 TLS 1.3 ID

Edwards / Montgomery curves:

Identifier	Required build option
WOLFSSL_ECC_X25519	HAVE_CURVE25519
WOLFSSL_ECC_X448	HAVE_CURVE448

SM2 (requires WOLFSSL_SM2):

Identifier	Group
WOLFSSL_ECC_SM2P256V1	sm2p256v1

Finite-field DH (RFC 7919) groups (require HAVE_FFDHE and the matching HAVE_FFDHE_NNNN macro for each size):

Identifier	Group
WOLFSSL_FFDHE_2048	ffdhe2048
WOLFSSL_FFDHE_3072	ffdhe3072
WOLFSSL_FFDHE_4096	ffdhe4096
WOLFSSL_FFDHE_6144	ffdhe6144
WOLFSSL_FFDHE_8192	ffdhe8192

ML-KEM (post-quantum) groups (require HAVE_PQC, WOLFSSL_HAVE_MLKEM and !WOLFSSL_NO_ML_KEM):

Identifier
WOLFSSL_ML_KEM_512
WOLFSSL_ML_KEM_768
WOLFSSL_ML_KEM_1024

ML-KEM hybrid groups additionally require HAVE_ECC together with either WOLFSSL_WC_MLKEM, and WOLFSSL_PQC_HYBRIDS (or WOLFSSL_EXTRA_PQC_HYBRIDS for the “extra” set):

Identifier	Hybrid flag set
WOLFSSL_SECP256R1MLKEM768	WOLFSSL_PQC_HYBRIDS

Identifier	Hybrid flag set
WOLFSSL_X25519MLKEM768	WOLFSSL_PQC_HYBRIDS
WOLFSSL_SECP384R1MLKEM1024	WOLFSSL_PQC_HYBRIDS
WOLFSSL_SECP256R1MLKEM512	WOLFSSL_EXTRA_PQC_HYBRIDS
WOLFSSL_SECP384R1MLKEM768	WOLFSSL_EXTRA_PQC_HYBRIDS
WOLFSSL_SECP521R1MLKEM1024	WOLFSSL_EXTRA_PQC_HYBRIDS
WOLFSSL_X25519MLKEM512	WOLFSSL_EXTRA_PQC_HYBRIDS
WOLFSSL_X448MLKEM768	WOLFSSL_EXTRA_PQC_HYBRIDS

Legacy Kyber groups (require HAVE_PQC and WOLFSSL_MLKEM_KYBER; hybrids additionally require HAVE_ECC together with WOLFSSL_WC_MLKEM):

Identifier
WOLFSSL_KYBER_LEVEL1
WOLFSSL_KYBER_LEVEL3
WOLFSSL_KYBER_LEVEL5
WOLFSSL_P256_KYBER_LEVEL1
WOLFSSL_P256_KYBER_LEVEL3
WOLFSSL_P384_KYBER_LEVEL3
WOLFSSL_P521_KYBER_LEVEL5
WOLFSSL_X25519_KYBER_LEVEL1
WOLFSSL_X25519_KYBER_LEVEL3
WOLFSSL_X448_KYBER_LEVEL3

Example

```
int ret;
WOLFSSL_CTX* ctx;
int groups[] = { WOLFSSL_ECC_X25519, WOLFSSL_ECC_SECP256R1 };
int count = sizeof(groups) / sizeof(groups[0]);
...
ret = wolfSSL_CTX_set_groups(ctx, groups, count);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set group list
}
```

C.58.4.471 function wolfSSL_set_groups

```
int wolfSSL_set_groups(
    WOLFSSL * ssl,
    int * groups,
    int count
)
```

This function sets the list of key-exchange groups (named elliptic curves and KEMs) to allow on a wolfSSL session in order of preference. The array format and the set of recognized identifiers are identical to those documented for wolfSSL_CTX_set_groups(); refer to that function for the full identifier table and required build options.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **groups** a list of key-exchange groups by identifier.

- **count** the number of identifiers in groups (must not exceed WOLFSSL_MAX_GROUP_COUNT).

See:

- wolfSSL_CTX_set_groups
- wolfSSL_UseKeyShare
- wolfSSL_CTX_set1_groups_list
- wolfSSL_set1_groups_list
- wolfSSL_preferred_group

Return:

- BAD_FUNC_ARG if a pointer parameter is NULL, count exceeds WOLFSSL_MAX_GROUP_COUNT, any of the identifiers are unrecognized, or the underlying method is not a TLS method.
- WOLFSSL_SUCCESS if successful.

Example

```
int ret;
WOLFSSL* ssl;
int groups[] = { WOLFSSL_ECC_X25519, WOLFSSL_ECC_SECP256R1 };
int count = sizeof(groups) / sizeof(groups[0]);
...
ret = wolfSSL_set_groups(ssl, groups, count);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set group list
}
```

C.58.4.472 function wolfSSL_connect_TLSv13

```
int wolfSSL_connect_TLSv13(
    WOLFSSL * ssl
)
```

This function is called on the client side and initiates a TLS v1.3 handshake with a server. When this function is called, the underlying communication channel has already been set up. wolfSSL_connect() works with both blocking and non_blocking I/O. When the underlying I/O is non_blocking, wolfSSL_connect() will return when the underlying I/O could not satisfy the needs of wolfSSL_connect to continue the handshake. In this case, a call to wolfSSL_get_error() will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_connect() when the underlying I/O is ready and wolfSSL will pick up where it left off. When using a non_blocking socket, nothing needs to be done, but select() can be used to check for the required condition. If the underlying I/O is blocking, wolfSSL_connect() will only return once the handshake has been finished or an error occurred. wolfSSL takes a different approach to certificate verification than OpenSSL does. The default policy for the client is to verify the server, this means that if you don't load CAs to verify the server you'll get a connect error, unable to verify (_155). It you want to mimic OpenSSL behavior of having SSL_connect succeed even if verifying the server fails and reducing security you can do this by calling: SSL_CTX_set_verify(ctx, SSL_VERIFY_NONE, 0); before calling SSL_new(); Though it's not recommended.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_get_error
- wolfSSL_connect
- wolfSSL_accept_TLSv13
- wolfSSL_accept

Return:

- SSL_SUCCESS upon success.
- SSL_FATAL_ERROR will be returned if an error occurred. To get a more detailed error code, call wolfSSL_get_error().

If decrypted application data is waiting to be read, the handshake is not continued and wolfSSL_get_error() returns APP_DATA_READY. Read the data with wolfSSL_read() and then call wolfSSL_connect_TLSv13() again.

Example

```
int ret = 0;
int err = 0;
WOLFSSL* ssl;
char buffer[80];
...

ret = wolfSSL_connect_TLSv13(ssl);
if (ret != SSL_SUCCESS) {
    err = wolfSSL_get_error(ssl, ret);
    printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
}
```

C.58.4.473 function wolfSSL_accept_TLSv13

```
wolfSSL_accept_TLSv13(
    WOLFSSL * ssl
)
```

This function is called on the server side and waits for a SSL/TLS client to initiate the SSL/TLS handshake. When this function is called, the underlying communication channel has already been set up. wolfSSL_accept() works with both blocking and non_blocking I/O. When the underlying I/O is non_blocking, wolfSSL_accept() will return when the underlying I/O could not satisfy the needs of wolfSSL_accept to continue the handshake. In this case, a call to wolfSSL_get_error() will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_accept when data is available to read and wolfSSL will pick up where it left off. When using a non_blocking socket, nothing needs to be done, but select() can be used to check for the required condition. If the underlying I/O is blocking, wolfSSL_accept() will only return once the handshake has been finished or an error occurred. Call this function when expecting a TLS v1.3 connection though older version ClientHello messages are supported.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_get_error
- wolfSSL_connect_TLSv13
- wolfSSL_connect
- wolfSSL_accept_TLSv13
- wolfSSL_accept

Return:

- SSL_SUCCESS upon success.
- SSL_FATAL_ERROR will be returned if an error occurred. To get a more detailed error code, call wolfSSL_get_error().

Example

```
int ret = 0;
int err = 0;
WOLFSSL* ssl;
char buffer[80];
...

ret = wolfSSL_accept_TLSv13(ssl);
if (ret != SSL_SUCCESS) {
    err = wolfSSL_get_error(ssl, ret);
    printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
}
```

C.58.4.474 function wolfSSL_CTX_set_max_early_data

```
int wolfSSL_CTX_set_max_early_data(
    WOLFSSL_CTX * ctx,
    unsigned int sz
)
```

This function sets the maximum amount of early data that a TLS v1.3 client or server is willing to exchange using the wolfSSL context. Call this function to limit the amount of early data to process to mitigate replay attacks. Early data is protected by keys derived from those of the connection that the session ticket was sent and therefore will be the same every time a session ticket is used in resumption. The value is included in the session ticket for resumption. A server value of zero indicates no early data is to be sent by client using session tickets. A client value of zero indicates that the client will not send any early data. The default value is zero: per RFC 8446 Appendix E.5, TLS implementations "MUST NOT enable 0-RTT (either sending or accepting) unless specifically requested by the application." Servers must call this function (or the per-SSL equivalent) with a non-zero value to opt in. It is recommended that the number of early data bytes be kept as low as practically possible in the application.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().
- **sz** the amount of early data to accept in bytes.

See:

- wolfSSL_set_max_early_data
- wolfSSL_write_early_data
- wolfSSL_read_early_data

Return:

- BAD_FUNC_ARG if ctx is NULL or not using TLS v1.3.
- 0 if successful.

Example

```
int ret;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_set_max_early_data(ctx, 128);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set group list
}
```

C.58.4.475 function wolfSSL_set_max_early_data

```
int wolfSSL_set_max_early_data(
    WOLFSSL * ssl,
    unsigned int sz
)
```

This function sets the maximum amount of early data that a TLS v1.3 client or server is willing to exchange. Call this function to limit the amount of early data to process to mitigate replay attacks. Early data is protected by keys derived from those of the connection that the session ticket was sent and therefore will be the same every time a session ticket is used in resumption. The value is included in the session ticket for resumption. A server value of zero indicates no early data is to be sent by client using session tickets. A client value of zero indicates that the client will not send any early data. It is recommended that the number of early data bytes be kept as low as practically possible in the application.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **sz** the amount of early data to accept from client in bytes.

See:

- wolfSSL_CTX_set_max_early_data
- wolfSSL_write_early_data
- wolfSSL_read_early_data

Return:

- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3.
- 0 if successful.

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_set_max_early_data(ssl, 128);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set group list
}
```

C.58.4.476 function wolfSSL_write_early_data

```
int wolfSSL_write_early_data(
    WOLFSSL * ssl,
    const void * data,
    int sz,
    int * outSz
)
```

This function writes early data to the server on resumption. Call this function before wolfSSL_connect() or wolfSSL_connect_TLSv13(). This function is only used with clients.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **data** the buffer holding the early data to write to server.
- **sz** the amount of early data to write in bytes.
- **outSz** the amount of early data written in bytes.

See:

- wolfSSL_read_early_data
- wolfSSL_connect
- wolfSSL_connect_TLSv13

Return:

- BAD_FUNC_ARG if a pointer parameter is NULL, sz is less than 0 or not using TLSv1.3.
- SIDE_ERROR if called with a server.
- BAD_STATE_E if invoked without a valid session or without a valid PSK cb
- WOLFSSL_FATAL_ERROR if the connection is not made, or if the AEAD key usage limit would be exceeded by this write, in which case wolfSSL_get_error() reports TOO_MUCH_EARLY_DATA. A KeyUpdate cannot be performed while sending early data (RFC 9846, Section 5.5), so no further early data can be sent on this connection. The write is all-or-nothing as usual: a value less than sz is never returned. Note that a failed call may still have put records on the wire before the limit was reached, so treat the connection as unusable for early data rather than resuming the send from an offset.
- the amount of early data written in bytes if successful.

Example

```
int ret = 0;
int err = 0;
WOLFSSL* ssl;
byte earlyData[] = { early data };
int outSz;
char buffer[80];
...

ret = wolfSSL_write_early_data(ssl, earlyData, sizeof(earlyData), &outSz);
if (ret < 0) {
    err = wolfSSL_get_error(ssl, ret);
    printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
    goto err_label;
}
if (outSz < sizeof(earlyData)) {
    // not all early data was sent
}
ret = wolfSSL_connect_TLSv13(ssl);
if (ret != SSL_SUCCESS) {
    err = wolfSSL_get_error(ssl, ret);
    printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
}
```

C.58.4.477 function wolfSSL_read_early_data

```
int wolfSSL_read_early_data(
    WOLFSSL * ssl,
    void * data,
    int sz,
    int * outSz
)
```

This function reads any early data from a client on resumption. Call this function instead of wolfSSL_accept() or wolfSSL_accept_TLSv13() to accept a client and read any early data in the handshake. The function should be invoked until wolfSSL_is_init_finished() returns true. Early data may be sent

by the client in multiple messages. If there is no early data then the handshake will be processed as normal. This function is only used with servers.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **data** a buffer to hold the early data read from client.
- **sz** size of the buffer in bytes.
- **outSz** number of bytes of early data read.

See:

- wolfSSL_write_early_data
- wolfSSL_accept
- wolfSSL_accept_TLSv13

Return:

- BAD_FUNC_ARG if a pointer parameter is NULL, sz is less than 0 or not using TLSv1.3.
- SIDE_ERROR if called with a client.
- WOLFSSL_FATAL_ERROR if accepting a connection fails.
- Number of early data bytes read (may be zero).

Example

```
int ret = 0;
int err = 0;
WOLFSSL* ssl;
byte earlyData[128];
int outSz;
char buffer[80];
...

do {
    ret = wolfSSL_read_early_data(ssl, earlyData, sizeof(earlyData), &outSz);
    if (ret < 0) {
        err = wolfSSL_get_error(ssl, ret);
        printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
    }
    if (outSz > 0) {
        // early data available
    }
} while (!wolfSSL_is_init_finished(ssl));
```

C.58.4.478 function wolfSSL_CTX_no_early_data_fresh_start_check

```
int wolfSSL_CTX_no_early_data_fresh_start_check(
    WOLFSSL_CTX * ctx
)
```

This function is called on the server to disable the RFC 8446 Section 8.2 fresh start protection. By default a freshly created context rejects early data, but not resumption, for session tickets minted before the context was created, since the anti-replay state for those tickets may not have survived a server restart. Only call this function when the anti-replay state (session cache or external cache) reliably survives server restarts.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().

See:

- wolfSSL_CTX_set_max_early_data
- wolfSSL_read_early_data

Return:

- BAD_FUNC_ARG if ctx is NULL or not using TLS v1.3.
- SIDE_ERROR if called with a client.
- 0 if successful.

The check compares the ticket timestamp against the context creation time, both taken from TimeNowInMilliseconds(). That clock is only required to be millisecond accurate, not correlated to the epoch, so on ports where it counts from boot (Windows QPC, Zephyr, FreeRTOS, Micrium, Microchip) it restarts near zero and the check does not fire for tickets minted before a reboot. Such deployments must rely on ticket key rotation instead.

Example

```
int ret;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_no_early_data_fresh_start_check(ctx);
if (ret != 0) {
    // failed to disable the fresh start check
}
```

C.58.4.479 function wolfSSL_inject

```
int wolfSSL_inject(
    WOLFSSL * ssl,
    const void * data,
    int sz
)
```

This function is called to inject data into the WOLFSSL object. This is useful when data needs to be read from a single place and demultiplexed into multiple connections. The caller should then call wolfSSL_read() to extract the plaintext data from the WOLFSSL object.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **data** data to inject into the ssl object.
- **sz** number of bytes of data to inject.

See: wolfSSL_read

Return:

- BAD_FUNC_ARG if any pointer parameter is NULL or sz <= 0
- BUFFER_ERROR if the input buffer lengths are inconsistent
- APP_DATA_READY if the input buffer must be grown while there is application data left to read
- MEMORY_E if allocation fails
- WOLFSSL_SUCCESS on success

Example

```
byte buf[2000]
sz = recv(fd, buf, sizeof(buf), 0);
if (sz <= 0)
    // error
```

```
if (wolfSSL_inject(ssl, buf, sz) != WOLFSSL_SUCCESS)
    // error
sz = wolfSSL_read(ssl, buf, sizeof(buf);
```

C.58.4.480 function wolfSSL_CTX_set_psk_client_tls13_callback

```
void wolfSSL_CTX_set_psk_client_tls13_callback(
    WOLFSSL_CTX * ctx,
    wc_psk_client_tls13_callback cb
)
```

This function sets the Pre-Shared Key (PSK) client side callback for TLS v1.3 connections. The callback is used to find a PSK identity and return its key and the name of the cipher to use for the handshake. The function sets the client_psk_tls13_cb member of the WOLFSSL_CTX structure.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().
- **cb** a Pre-Shared Key (PSK) callback for a TLS 1.3 client.

See:

- wolfSSL_set_psk_client_tls13_callback
- wolfSSL_CTX_set_psk_server_tls13_callback
- wolfSSL_set_psk_server_tls13_callback

Example

```
WOLFSSL_CTX* ctx;
...
wolfSSL_CTX_set_psk_client_tls13_callback(ctx, my_psk_client_tls13_cb);
```

C.58.4.481 function wolfSSL_set_psk_client_tls13_callback

```
void wolfSSL_set_psk_client_tls13_callback(
    WOLFSSL * ssl,
    wc_psk_client_tls13_callback cb
)
```

This function sets the Pre-Shared Key (PSK) client side callback for TLS v1.3 connections. The callback is used to find a PSK identity and return its key and the name of the cipher to use for the handshake. The function sets the client_psk_tls13_cb member of the options field in WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **cb** a Pre-Shared Key (PSK) callback for a TLS 1.3 client.

See:

- wolfSSL_CTX_set_psk_client_tls13_callback
- wolfSSL_CTX_set_psk_server_tls13_callback
- wolfSSL_set_psk_server_tls13_callback

Example

```
WOLFSSL* ssl;
...
wolfSSL_set_psk_client_tls13_callback(ssl, my_psk_client_tls13_cb);
```

C.58.4.482 function wolfSSL_CTX_set_psk_server_tls13_callback

```
void wolfSSL_CTX_set_psk_server_tls13_callback(
    WOLFSSL_CTX * ctx,
    wc_psk_server_tls13_callback cb
)
```

This function sets the Pre-Shared Key (PSK) server side callback for TLS v1.3 connections. The callback is used to find a PSK identity and return its key and the name of the cipher to use for the handshake. The function sets the server_psk_tls13_cb member of the WOLFSSL_CTX structure.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().
- **cb** a Pre-Shared Key (PSK) callback for a TLS 1.3 server.

See:

- wolfSSL_CTX_set_psk_client_tls13_callback
- wolfSSL_set_psk_client_tls13_callback
- wolfSSL_set_psk_server_tls13_callback

Example

```
WOLFSSL_CTX* ctx;
...
wolfSSL_CTX_set_psk_server_tls13_callback(ctx, my_psk_client_tls13_cb);
```

C.58.4.483 function wolfSSL_set_psk_server_tls13_callback

```
void wolfSSL_set_psk_server_tls13_callback(
    WOLFSSL * ssl,
    wc_psk_server_tls13_callback cb
)
```

This function sets the Pre-Shared Key (PSK) server side callback for TLS v1.3 connections. The callback is used to find a PSK identity and return its key and the name of the cipher to use for the handshake. The function sets the server_psk_tls13_cb member of the options field in WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **cb** a Pre-Shared Key (PSK) callback for a TLS 1.3 server.

See:

- wolfSSL_CTX_set_psk_client_tls13_callback
- wolfSSL_set_psk_client_tls13_callback
- wolfSSL_CTX_set_psk_server_tls13_callback

Example

```
WOLFSSL* ssl;
...
wolfSSL_set_psk_server_tls13_callback(ssl, my_psk_server_tls13_cb);
```

C.58.4.484 function wolfSSL_CTX_set_cert_with_extern_psk

```
int wolfSSL_CTX_set_cert_with_extern_psk(
    WOLFSSL_CTX * ctx,
    int state
)
```

Enable or disable TLS 1.3 certificate authentication with external PSK (RFC 9973) on a context.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().
- **state** 0 to disable, non-zero to enable.

See:

- wolfSSL_set_cert_with_extern_psk
- wolfSSL_CTX_set_psk_client_tls13_callback
- wolfSSL_CTX_set_psk_server_tls13_callback

Return:

- WOLFSSL_SUCCESS on success.
- WOLFSSL_FAILURE when `ctx` is NULL.

When enabled, wolfSSL advertises and accepts the `tls_cert_with_extern_psk` extension for TLS 1.3 handshakes using external PSKs. Any non-zero `state` value enables the feature and zero disables it.

Availability:

- Built with `--enable-tls13 --enable-psk --enable-cert-with-extern-psk`
- Or with `WOLFSSL_TLS13` and `WOLFSSL_CERT_WITH_EXTERN_PSK` defined

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(wolfTLSv1_3_client_method());
if (wolfSSL_CTX_set_cert_with_extern_psk(ctx, 1) != WOLFSSL_SUCCESS) {
    /* handle error */
}
```

C.58.4.485 function wolfSSL_set_cert_with_extern_psk

```
int wolfSSL_set_cert_with_extern_psk(
    WOLFSSL * ssl,
    int state
)
```

Enable or disable TLS 1.3 certificate authentication with external PSK (RFC 9973) on a connection.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **state** 0 to disable, non-zero to enable.

See:

- wolfSSL_CTX_set_cert_with_extern_psk
- wolfSSL_set_psk_client_tls13_callback
- wolfSSL_set_psk_server_tls13_callback

Return:

- WOLFSSL_SUCCESS on success.
- WOLFSSL_FAILURE when `ssl` is NULL.

This call applies to a single WOLFSSL object. Any non-zero `state` value enables the feature and zero disables it.

Availability:

- Built with `--enable-tls13 --enable-psk --enable-cert-with-extern-psk`

- Or with WOLFSSL_TLS13 and WOLFSSL_CERT_WITH_EXTERN_PSK defined

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
if (wolfSSL_set_cert_with_extern_psk(ssl, 1) != WOLFSSL_SUCCESS) {
    /* handle error */
}
```

C.58.4.486 function wolfSSL_UseKeyShare

```
int wolfSSL_UseKeyShare(
    WOLFSSL * ssl,
    word16 group
)
```

This function creates a key share entry from the group including generating a key pair. The KeyShare extension contains all the generated public keys for key exchange. If this function is called, then only the groups specified will be included. Call this function when a preferred group has been previously established for the server.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **group** a key exchange group identifier.

See:

- wolfSSL_preferred_group
- wolfSSL_CTX_set1_groups_list
- wolfSSL_set1_groups_list
- wolfSSL_CTX_set_groups
- wolfSSL_set_groups
- wolfSSL_NoKeyShares

Return:

- BAD_FUNC_ARG if ssl is NULL.
- MEMORY_E when dynamic memory allocation fails.
- WOLFSSL_SUCCESS if successful.

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_UseKeyShare(ssl, WOLFSSL_ECC_X25519);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set key share
}
```

C.58.4.487 function wolfSSL_NoKeyShares

```
int wolfSSL_NoKeyShares(
    WOLFSSL * ssl
)
```

This function is called to ensure no key shares are sent in the ClientHello. This will force the server to respond with a HelloRetryRequest if a key exchange is required in the handshake. Call this function when the expected key exchange group is not known and to avoid the generation of keys unnecessarily.

Note that an extra round-trip will be required to complete the handshake when a key exchange is required.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_UseKeyShare

Return:

- BAD_FUNC_ARG if ssl is NULL.
- SIDE_ERROR if called with a server.
- WOLFSSL_SUCCESS if successful.

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_NoKeyShares(ssl);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set no key shares
}
```

C.58.4.488 function wolfTLSv1_3_server_method_ex

```
WOLFSSL_METHOD * wolfTLSv1_3_server_method_ex(
    void * heap
)
```

This function is used to indicate that the application is a server and will only support the TLS 1.3 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Parameters:

- **heap** a pointer to a buffer that the static memory allocator will use during dynamic memory allocation.

See:

- wolfSSLv3_server_method
- wolfTLSv1_server_method
- wolfTLSv1_1_server_method
- wolfTLSv1_2_server_method
- wolfTLSv1_3_server_method
- wolfDTLSv1_server_method
- wolfSSLv23_server_method
- wolfSSL_CTX_new

Return:

- If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
```

```
WOLFSSL_CTX* ctx;

method = wolfTLSv1_3_server_method_ex(NULL);
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.58.4.489 function wolfTLSv1_3_client_method_ex

```
WOLFSSL_METHOD * wolfTLSv1_3_client_method_ex(
    void * heap
)
```

This function is used to indicate that the application is a client and will only support the TLS 1.3 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Parameters:

- **heap** a pointer to a buffer that the static memory allocator will use during dynamic memory allocation.

See:

- wolfSSLv3_client_method
- wolfTLSv1_client_method
- wolfTLSv1_1_client_method
- wolfTLSv1_2_client_method
- wolfTLSv1_3_client_method
- wolfDTLSv1_client_method
- wolfSSLv23_client_method
- wolfSSL_CTX_new

Return:

- If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_3_client_method_ex(NULL);
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.58.4.490 function wolfTLSv1_3_server_method

```
WOLFSSL_METHOD * wolfTLSv1_3_server_method(
    void
)
```

This function is used to indicate that the application is a server and will only support the TLS 1.3 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

See:

- wolfSSLv3_server_method
- wolfTLSv1_server_method
- wolfTLSv1_1_server_method
- wolfTLSv1_2_server_method
- wolfTLSv1_3_server_method_ex
- wolfDTLSv1_server_method
- wolfSSLv23_server_method
- wolfSSL_CTX_new

Return:

- If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_3_server_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.58.4.491 function wolfTLSv1_3_client_method

```
WOLFSSL_METHOD * wolfTLSv1_3_client_method(
    void
)
```

This function is used to indicate that the application is a client and will only support the TLS 1.3 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

See:

- wolfSSLv3_client_method
- wolfTLSv1_client_method
- wolfTLSv1_1_client_method
- wolfTLSv1_2_client_method
- wolfTLSv1_3_client_method_ex
- wolfDTLSv1_client_method

- wolfSSLv23_client_method
- wolfSSL_CTX_new

Return:

- If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_3_client_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.58.4.492 function wolfTLSv1_3_method_ex

```
WOLFSSL_METHOD * wolfTLSv1_3_method_ex(
    void * heap
)
```

This function returns a WOLFSSL_METHOD similar to wolfTLSv1_3_client_method except that it is not determined which side yet (server/client).

Parameters:

- **heap** a pointer to a buffer that the static memory allocator will use during dynamic memory allocation.

See:

- wolfSSL_new
- wolfSSL_free

Return:

- WOLFSSL_METHOD On successful creations returns a WOLFSSL_METHOD pointer
- NULL Null if memory allocation error or failure to create method

Example

```
WOLFSSL* ctx;
ctx  = wolfSSL_CTX_new(wolfTLSv1_3_method_ex(NULL));
// check ret value
```

C.58.4.493 function wolfTLSv1_3_method

```
WOLFSSL_METHOD * wolfTLSv1_3_method(
    void
)
```

This function returns a WOLFSSL_METHOD similar to wolfTLSv1_3_client_method except that it is not determined which side yet (server/client).

See:

- wolfSSL_new
- wolfSSL_free

Return:

- WOLFSSL_METHOD On successful creations returns a WOLFSSL_METHOD pointer
- NULL Null if memory allocation error or failure to create method

Example

```
WOLFSSL* ctx;
ctx  = wolfSSL_CTX_new(wolfTLSv1_3_method());
// check ret value
```

C.58.4.494 function wolfSSL_CTX_set_ephemeral_key

```
int wolfSSL_CTX_set_ephemeral_key(
    WOLFSSL_CTX * ctx,
    int keyAlgo,
    const char * key,
    unsigned int keySz,
    int format
)
```

This function sets a fixed / static ephemeral key for testing only.

Parameters:

- **ctx** A WOLFSSL_CTX context pointer
- **keyAlgo** enum wc_PkType like WC_PK_TYPE_DH and WC_PK_TYPE_ECDH
- **key** key file path (if keySz == 0) or actual key buffer (PEM or ASN.1)
- **keySz** key size (should be 0 for “key” arg is file path)
- **format** WOLFSSL_FILETYPE_ASN1 or WOLFSSL_FILETYPE_PEM

See: wolfSSL_CTX_get_ephemeral_key

Return: 0 Key loaded successfully

C.58.4.495 function wolfSSL_set_ephemeral_key

```
int wolfSSL_set_ephemeral_key(
    WOLFSSL * ssl,
    int keyAlgo,
    const char * key,
    unsigned int keySz,
    int format
)
```

This function sets a fixed / static ephemeral key for testing only.

Parameters:

- **ssl** A WOLFSSL object pointer
- **keyAlgo** enum wc_PkType like WC_PK_TYPE_DH and WC_PK_TYPE_ECDH
- **key** key file path (if keySz == 0) or actual key buffer (PEM or ASN.1)
- **keySz** key size (should be 0 for “key” arg is file path)

- **format** WOLFSSL_FILETYPE_ASN1 or WOLFSSL_FILETYPE_PEM

See: wolfSSL_get_ephemeral_key

Return: 0 Key loaded successfully

C.58.4.496 function wolfSSL_CTX_get_ephemeral_key

```
int wolfSSL_CTX_get_ephemeral_key(
    WOLFSSL_CTX * ctx,
    int keyAlgo,
    const unsigned char ** key,
    unsigned int * keySz
)
```

This function returns pointer to loaded key as ASN.1/DER.

Parameters:

- **ctx** A WOLFSSL_CTX context pointer
- **keyAlgo** enum wc_PkType like WC_PK_TYPE_DH and WC_PK_TYPE_ECDH
- **key** key buffer pointer
- **keySz** key size pointer

See: wolfSSL_CTX_set_ephemeral_key

Return: 0 Key returned successfully

C.58.4.497 function wolfSSL_get_ephemeral_key

```
int wolfSSL_get_ephemeral_key(
    WOLFSSL * ssl,
    int keyAlgo,
    const unsigned char ** key,
    unsigned int * keySz
)
```

This function returns pointer to loaded key as ASN.1/DER.

Parameters:

- **ssl** A WOLFSSL object pointer
- **keyAlgo** enum wc_PkType like WC_PK_TYPE_DH and WC_PK_TYPE_ECDH
- **key** key buffer pointer
- **keySz** key size pointer

See: wolfSSL_set_ephemeral_key

Return: 0 Key returned successfully

C.58.4.498 function wolfSSL_RSA_sign_generic_padding

```
int wolfSSL_RSA_sign_generic_padding(
    int hashAlg,
    const unsigned char * hash,
    unsigned int hLen,
    unsigned char * sigRet,
    unsigned int * sigLen,
    WOLFSSL_RSA * rsa,
    int flag,
```

```
    int padding
)
```

Sign a message with the chosen message digest, padding, and RSA key.

Parameters:

- **type** Hash NID
- **m** Message to sign. Most likely this will be the digest of the message to sign
- **mLen** Length of message to sign
- **sigRet** Output buffer
- **sigLen** On Input: length of sigRet buffer On Output: length of data written to sigRet
- **rsa** RSA key used to sign the input
- **flag** 1: Output the signature 0: Output the value that the unpadded signature should be compared to. Note: for RSA_PKCS1_PSS_PADDING the wc_RsaPSS_CheckPadding_ex function should be used to check the output of a *Verify* function.
- **padding** Padding to use. Only RSA_PKCS1_PSS_PADDING and RSA_PKCS1_PADDING are currently supported for signing.

Return: WOLFSSL_SUCCESS on success and c on error

C.58.4.499 function wolfSSL_dtls13_has_pending_msg

```
int wolfSSL_dtls13_has_pending_msg(
    WOLFSSL * ssl
)
```

checks if DTLSv1.3 stack has some messages sent but not yet acknowledged by the other peer

Parameters:

- **ssl** A WOLFSSL object pointer

Return: 1 if there are pending messages, 0 otherwise

C.58.4.500 function wolfSSL_SESSION_get_max_early_data

```
unsigned int wolfSSL_SESSION_get_max_early_data(
    const WOLFSSL_SESSION * s
)
```

Get the maximum size of Early Data from a session.

Parameters:

- **s** the WOLFSSL_SESSION instance.

See:

- wolfSSL_set_max_early_data
- wolfSSL_write_early_data
- wolfSSL_read_early_data

Return: the value of max_early_data that was configured in the WOLFSSL* the session was derived from.

C.58.4.501 function wolfSSL_CRYPTO_get_ex_new_index

```
int wolfSSL_CRYPTO_get_ex_new_index(
    int class_index,
    long argl,
```

```
    void * argp,
    WOLFSSL_CRYPTO_EX_new * new_func,
    WOLFSSL_CRYPTO_EX_dup * dup_func,
    WOLFSSL_CRYPTO_EX_free * free_func
)
```

Get a new index for external data. This entry applies also for the following API:

Parameters:

- **class_index** Identifier for the object class the external data index applies to. Ignored by wolfSSL.
- **argl** Optional long argument passed through for compatibility. Ignored by wolfSSL.
- **argp** Optional pointer argument passed through for compatibility. Ignored by wolfSSL.
- **new_func** Pointer to an external data constructor callback. Ignored by wolfSSL.
- **dup_func** Pointer to an external data duplicate callback. Ignored by wolfSSL.
- **free_func** Pointer to an external data destructor callback. Ignored by wolfSSL.

Return: The new index value to be used with the external data API for this object class.

- wolfSSL_CTX_get_ex_new_index
- wolfSSL_get_ex_new_index
- wolfSSL_SESSION_get_ex_new_index
- wolfSSL_X509_get_ex_new_index

C.58.4.502 function wolfSSL_CTX_set_client_cert_type

```
int wolfSSL_CTX_set_client_cert_type(
    WOLFSSL_CTX * ctx,
    const char * buf,
    int len
)
```

In case this function is called in a client side, set certificate types that can be sent to its peer. In case called in a server side, set certificate types that can be acceptable from its peer. Put cert types in the buffer with prioritised order. To reset the settings to default, pass NULL for the buffer or pass zero for len. By default, certificate type is only X509. In case both side intend to send or accept “Raw public key” cert, WOLFSSL_CERT_TYPE_RPK should be included in the buffer to set.

Parameters:

- **ctx** WOLFSSL_CTX object pointer
- **buf** A buffer where certificate types are stored
- **len** buf size in bytes (same as number of certificate types included) *Example*

```
int ret;
WOLFSSL_CTX* ctx;
char buf[] = {WOLFSSL_CERT_TYPE_RPK, WOLFSSL_CERT_TYPE_X509};
int len = sizeof(buf)/sizeof(char);
...

ret = wolfSSL_CTX_set_client_cert_type(ctx, buf, len);
```

See:

- wolfSSL_set_client_cert_type
- wolfSSL_CTX_set_server_cert_type
- wolfSSL_set_server_cert_type
- wolfSSL_get_negotiated_client_cert_type
- wolfSSL_get_negotiated_server_cert_type

Return:

- WOLFSSL_SUCCESS if cert types set successfully
- BAD_FUNC_ARG if NULL was passed for ctx, illegal value was specified as cert type, buf size exceed MAX_CLIENT_CERT_TYPE_CNT was specified or a duplicate value is found in buf.

C.58.4.503 function wolfSSL_CTX_set_server_cert_type

```
int wolfSSL_CTX_set_server_cert_type(
    WOLFSSL_CTX * ctx,
    const char * buf,
    int len
)
```

In case this function is called in a server side, set certificate types that can be sent to its peer. In case called in a client side, set certificate types that can be acceptable from its peer. Put cert types in the buffer with prioritised order. To reset the settings to default, pass NULL for the buffer or pass zero for len. By default, certificate type is only X509. In case both side intend to send or accept "Raw public key" cert, WOLFSSL_CERT_TYPE_RPK should be included in the buffer to set.

Parameters:

- **ctx** WOLFSSL_CTX object pointer
- **buf** A buffer where certificate types are stored
- **len** buf size in bytes (same as number of certificate types included) *Example*

```
int ret;
WOLFSSL_CTX* ctx;
char buf[] = {WOLFSSL_CERT_TYPE_RPK, WOLFSSL_CERT_TYPE_X509};
int len = sizeof(buf)/sizeof(char);
...

ret = wolfSSL_CTX_set_server_cert_type(ctx, buf, len);
```

See:

- wolfSSL_set_client_cert_type
- wolfSSL_CTX_set_client_cert_type
- wolfSSL_set_server_cert_type
- wolfSSL_get_negotiated_client_cert_type
- wolfSSL_get_negotiated_server_cert_type

Return:

- WOLFSSL_SUCCESS if cert types set successfully
- BAD_FUNC_ARG if NULL was passed for ctx, illegal value was specified as cert type, buf size exceed MAX_SERVER_CERT_TYPE_CNT was specified or a duplicate value is found in buf.

C.58.4.504 function wolfSSL_set_client_cert_type

```
int wolfSSL_set_client_cert_type(
    WOLFSSL * ssl,
    const char * buf,
    int len
)
```

In case this function is called in a client side, set certificate types that can be sent to its peer. In case called in a server side, set certificate types that can be acceptable from its peer. Put cert types in the buffer with prioritised order. To reset the settings to default, pass NULL for the buffer or pass zero for

len. By default, certificate type is only X509. In case both side intend to send or accept “Raw public key” cert, WOLFSSL_CERT_TYPE_RPK should be included in the buffer to set.

Parameters:

- **ssl** WOLFSSL object pointer
- **buf** A buffer where certificate types are stored
- **len** buf size in bytes (same as number of certificate types included) *Example*

```
int ret;
WOLFSSL* ssl;
char buf[] = {WOLFSSL_CERT_TYPE_RPK, WOLFSSL_CERT_TYPE_X509};
int len = sizeof(buf)/sizeof(char);
...

ret = wolfSSL_set_client_cert_type(ssl, buf, len);
```

See:

- wolfSSL_CTX_set_client_cert_type
- wolfSSL_CTX_set_server_cert_type
- wolfSSL_set_server_cert_type
- wolfSSL_get_negotiated_client_cert_type
- wolfSSL_get_negotiated_server_cert_type

Return:

- WOLFSSL_SUCCESS if cert types set successfully
- BAD_FUNC_ARG if NULL was passed for ctx, illegal value was specified as cert type, buf size exceed MAX_CLIENT_CERT_TYPE_CNT was specified or a duplicate value is found in buf.

C.58.4.505 function wolfSSL_set_server_cert_type

```
int wolfSSL_set_server_cert_type(
    WOLFSSL * ssl,
    const char * buf,
    int len
)
```

In case this function is called in a server side, set certificate types that can be sent to its peer. In case called in a client side, set certificate types that can be acceptable from its peer. Put cert types in the buffer with prioritised order. To reset the settings to default, pass NULL for the buffer or pass zero for len. By default, certificate type is only X509. In case both side intend to send or accept “Raw public key” cert, WOLFSSL_CERT_TYPE_RPK should be included in the buffer to set.

Parameters:

- **ctx** WOLFSSL_CTX object pointer
- **buf** A buffer where certificate types are stored
- **len** buf size in bytes (same as number of certificate types included) *Example*

```
int ret;
WOLFSSL* ssl;
char buf[] = {WOLFSSL_CERT_TYPE_RPK, WOLFSSL_CERT_TYPE_X509};
int len = sizeof(buf)/sizeof(char);
...

ret = wolfSSL_set_server_cert_type(ssl, buf, len);
```

See:

- wolfSSL_set_client_cert_type
- wolfSSL_CTX_set_server_cert_type
- wolfSSL_set_server_cert_type
- wolfSSL_get_negotiated_client_cert_type
- wolfSSL_get_negotiated_server_cert_type

Return:

- WOLFSSL_SUCCESS if cert types set successfully
- BAD_FUNC_ARG if NULL was passed for ctx, illegal value was specified as cert type, buf size exceed MAX_SERVER_CERT_TYPE_CNT was specified or a duplicate value is found in buf.

C.58.4.506 function wolfSSL_CTX_clear_group_messages

```
int wolfSSL_CTX_clear_group_messages(
    WOLFSSL_CTX * ctx
)
```

Disables handshake message grouping for the given WOLFSSL_CTX context.

Parameters:

- **ctx** Pointer to the WOLFSSL_CTX structure.

See:

- wolfSSL_CTX_set_group_messages
- wolfSSL_set_group_messages
- wolfSSL_clear_group_messages

Return:

- WOLFSSL_SUCCESS on success.
- BAD_FUNC_ARG if ctx is NULL.

This function turns off handshake message grouping for all SSL objects created from the specified context.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(wolfTLSv1_2_client_method());
wolfSSL_CTX_clear_group_messages(ctx);
```

C.58.4.507 function wolfSSL_clear_group_messages

```
int wolfSSL_clear_group_messages(
    WOLFSSL * ssl
)
```

Disables handshake message grouping for the given WOLFSSL object.

Parameters:

- **ssl** Pointer to the WOLFSSL structure.

See:

- wolfSSL_set_group_messages
- wolfSSL_CTX_set_group_messages
- wolfSSL_CTX_clear_group_messages

Return:

- WOLFSSL_SUCCESS on success.

- BAD_FUNC_ARG if ssl is NULL.

This function turns off handshake message grouping for the specified SSL object.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
wolfSSL_clear_group_messages(ssl);
```

C.58.4.508 function wolfSSL_get_negotiated_client_cert_type

```
int wolfSSL_get_negotiated_client_cert_type(
    WOLFSSL * ssl,
    int * tp
)
```

This function returns the result of the client certificate type negotiation done in ClientHello and ServerHello. WOLFSSL_SUCCESS is returned as a return value if no negotiation occurs and WOLFSSL_CERT_TYPE_UNKNOWN is returned as the certificate type.

Parameters:

- **ssl** WOLFSSL object pointer
- **tp** A buffer where a certificate type is to be returned. One of three certificate types will be returned: WOLFSSL_CERT_TYPE_RPK, WOLFSSL_CERT_TYPE_X509 or WOLFSSL_CERT_TYPE_UNKNOWN.

See:

- wolfSSL_set_client_cert_type
- wolfSSL_CTX_set_client_cert_type
- wolfSSL_set_server_cert_type
- wolfSSL_CTX_set_server_cert_type
- wolfSSL_get_negotiated_server_cert_type

Return:

- WOLFSSL_SUCCESS if a negotiated certificate type could be got
- BAD_FUNC_ARG if NULL was passed for ctx or tp

Example

```
int ret;
WOLFSSL* ssl;
int tp;
...

ret = wolfSSL_get_negotiated_client_cert_type(ssl, &tp);
```

C.58.4.509 function wolfSSL_get_negotiated_server_cert_type

```
int wolfSSL_get_negotiated_server_cert_type(
    WOLFSSL * ssl,
    int * tp
)
```

This function returns the result of the server certificate type negotiation done in ClientHello and ServerHello. WOLFSSL_SUCCESS is returned as a return value if no negotiation occurs and WOLFSSL_CERT_TYPE_UNKNOWN is returned as the certificate type.

Parameters:

- **ssl** WOLFSSL object pointer

- **tp** A buffer where a certificate type is to be returned. One of three certificate types will be returned: WOLFSSL_CERT_TYPE_RPK, WOLFSSL_CERT_TYPE_X509 or WOLFSSL_CERT_TYPE_UNKNOWN. *Example*

```
int ret;
WOLFSSL* ssl;
int tp;
...

ret = wolfSSL_get_negotiated_server_cert_type(ssl, &tp);
```

See:

- wolfSSL_set_client_cert_type
- wolfSSL_CTX_set_client_cert_type
- wolfSSL_set_server_cert_type
- wolfSSL_CTX_set_server_cert_type
- wolfSSL_get_negotiated_client_cert_type

Return:

- WOLFSSL_SUCCESS if a negotiated certificate type could be got
- BAD_FUNC_ARG if NULL was passed for ctx or tp

C.58.4.510 function wolfSSL_CTX_set_expected_rpk

```
int wolfSSL_CTX_set_expected_rpk(
    WOLFSSL_CTX * ctx,
    const unsigned char * spki,
    unsigned int spkiSz
)
```

Pin a DER-encoded SubjectPublicKeyInfo that the peer is expected to present as a Raw Public Key (RFC 7250), establishing out-of-band trust on the WOLFSSL_CTX object. An unauthenticated RPK peer is otherwise rejected: when the peer is being authenticated (any verify mode other than WOLFSSL_VERIFY_NONE) the handshake fails closed unless the presented key matches a pin (or a verify callback accepts it). May be called more than once to pin several keys, up to WOLFSSL_MAX_RPK_PINS. The key is stored as its SHA-256 digest, so this API requires SHA-256. Pins are append-only - there is no per-entry remove, but wolfSSL_CTX_clear_expected_rpk() empties the table so it can be repopulated.

Parameters:

- **ctx** WOLFSSL_CTX object pointer
- **spki** DER-encoded SubjectPublicKeyInfo the peer is expected to present
- **spkiSz** length of spki in bytes *Example*

```
int ret;
WOLFSSL_CTX* ctx;
const unsigned char* spki;
unsigned int spkiSz;
...

ret = wolfSSL_CTX_set_expected_rpk(ctx, spki, spkiSz);
```

See:

- wolfSSL_set_expected_rpk
- wolfSSL_CTX_clear_expected_rpk
- wolfSSL_set_client_cert_type
- wolfSSL_set_server_cert_type

Return:

- WOLFSSL_SUCCESS on success
- BAD_FUNC_ARG if ctx or spki is NULL, or spkiSz is 0
- BUFFER_E if the pin table is already full (WOLFSSL_MAX_RPK_PINS)
- other negative error code on a hashing failure

C.58.4.511 function wolfSSL_set_expected_rpk

```
int wolfSSL_set_expected_rpk(
    WOLFSSL * ssl,
    const unsigned char * spki,
    unsigned int spkiSz
)
```

Pin a DER-encoded SubjectPublicKeyInfo that the peer is expected to present as a Raw Public Key (RFC 7250), establishing out-of-band trust on the WOLFSSL object. An unauthenticated RPK peer is otherwise rejected: when the peer is being authenticated (any verify mode other than WOLFSSL_VERIFY_NONE) the handshake fails closed unless the presented key matches a pin (or a verify callback accepts it). May be called more than once to pin several keys, up to WOLFSSL_MAX_RPK_PINS. The key is stored as its SHA-256 digest, so this API requires SHA-256. Pins are append-only - there is no per-entry remove, but wolfSSL_clear_expected_rpk() empties the table so it can be repopulated.

Parameters:

- **ssl** WOLFSSL object pointer
- **spki** DER-encoded SubjectPublicKeyInfo the peer is expected to present
- **spkiSz** length of spki in bytes *Example*

```
int ret;
WOLFSSL* ssl;
const unsigned char* spki;
unsigned int spkiSz;
...

ret = wolfSSL_set_expected_rpk(ssl, spki, spkiSz);
```

See:

- wolfSSL_CTX_set_expected_rpk
- wolfSSL_clear_expected_rpk
- wolfSSL_set_client_cert_type
- wolfSSL_set_server_cert_type

Return:

- WOLFSSL_SUCCESS on success
- BAD_FUNC_ARG if ssl or spki is NULL, or spkiSz is 0
- BUFFER_E if the pin table is already full (WOLFSSL_MAX_RPK_PINS)
- other negative error code on a hashing failure

C.58.4.512 function wolfSSL_CTX_clear_expected_rpk

```
int wolfSSL_CTX_clear_expected_rpk(
    WOLFSSL_CTX * ctx
)
```

Remove all pinned expected peer Raw Public Keys (RFC 7250) from the WOLFSSL_CTX object, emptying the table so it can be repopulated - for example across a peer key rotation, since the pinning APIs are

otherwise append-only. Like other WOLFSSL_CTX setters this is not locked, so reconfigure the pins on a shared CTX while no handshakes are in flight (a WOLFSSL created with wolfSSL_new() copies the pin table by value at creation time).

Parameters:

- **ctx** WOLFSSL_CTX object pointer *Example*

```
WOLFSSL_CTX* ctx;
...

wolfSSL_CTX_clear_expected_rpk(ctx);
```

See:

- wolfSSL_CTX_set_expected_rpk
- wolfSSL_clear_expected_rpk

Return:

- WOLFSSL_SUCCESS on success
- BAD_FUNC_ARG if ctx is NULL

C.58.4.513 function wolfSSL_clear_expected_rpk

```
int wolfSSL_clear_expected_rpk(
    WOLFSSL * ssl
)
```

Remove all pinned expected peer Raw Public Keys (RFC 7250) from the WOLFSSL object, emptying the table so it can be repopulated - for example across a peer key rotation, since the pinning APIs are otherwise append-only.

Parameters:

- **ssl** WOLFSSL object pointer *Example*

```
WOLFSSL* ssl;
...

wolfSSL_clear_expected_rpk(ssl);
```

See:

- wolfSSL_set_expected_rpk
- wolfSSL_CTX_clear_expected_rpk

Return:

- WOLFSSL_SUCCESS on success
- BAD_FUNC_ARG if ssl is NULL

C.58.4.514 function wolfSSL_dtls_cid_use

```
int wolfSSL_dtls_cid_use(
    WOLFSSL * ssl
)
```

Enable use of ConnectionID extensions for the SSL object. See RFC 9146 and RFC 9147.

Parameters:

- **ssl** A WOLFSSL object pointer

See:

- wolfSSL_dtls_cid_is_enabled
- wolfSSL_dtls_cid_set
- wolfSSL_dtls_cid_get_rx_size
- wolfSSL_dtls_cid_get_rx
- wolfSSL_dtls_cid_get_tx_size
- wolfSSL_dtls_cid_get_tx

Return: WOLFSSL_SUCCESS on success, error code otherwise

C.58.4.515 function wolfSSL_dtls_cid_is_enabled

```
int wolfSSL_dtls_cid_is_enabled(
    WOLFSSL * ssl
)
```

If invoked after the handshake is complete it checks if ConnectionID was successfully negotiated for the SSL object. See RFC 9146 and RFC 9147.

Parameters:

- **ssl** A WOLFSSL object pointer

See:

- wolfSSL_dtls_cid_use
- wolfSSL_dtls_cid_set
- wolfSSL_dtls_cid_get_rx_size
- wolfSSL_dtls_cid_get_rx
- wolfSSL_dtls_cid_get_tx_size
- wolfSSL_dtls_cid_get_tx

Return: 1 if ConnectionID was correctly negotiated, 0 otherwise

C.58.4.516 function wolfSSL_dtls_cid_set

```
int wolfSSL_dtls_cid_set(
    WOLFSSL * ssl,
    unsigned char * cid,
    unsigned int size
)
```

Set the ConnectionID used by the other peer to send records in this connection. See RFC 9146 and RFC 9147. The ConnectionID must be at maximum DTLS_CID_MAX_SIZE, that is an tunable compile time define, and it can't never be bigger than 255 bytes.

Parameters:

- **ssl** A WOLFSSL object pointern
- **cid** the ConnectionID to be used
- **size** of the ConnectionID provided

See:

- wolfSSL_dtls_cid_use
- wolfSSL_dtls_cid_is_enabled
- wolfSSL_dtls_cid_get_rx_size
- wolfSSL_dtls_cid_get_rx
- wolfSSL_dtls_cid_get_tx_size
- wolfSSL_dtls_cid_get_tx

Return: WOLFSSL_SUCCESS if ConnectionID was correctly set, error code otherwise

C.58.4.517 function wolfSSL_dtls_cid_get_rx_size

```
int wolfSSL_dtls_cid_get_rx_size(
    WOLFSSL * ssl,
    unsigned int * size
)
```

Get the size of the ConnectionID used by the other peer to send records in this connection. See RFC 9146 and RFC 9147. The size is stored in the parameter size.

Parameters:

- **ssl** A WOLFSSL object pointern
- **size** a pointer to an unsigned int where the size will be stored

See:

- wolfSSL_dtls_cid_use
- wolfSSL_dtls_cid_is_enabled
- wolfSSL_dtls_cid_set
- wolfSSL_dtls_cid_get_rx
- wolfSSL_dtls_cid_get_tx_size
- wolfSSL_dtls_cid_get_tx

Return: WOLFSSL_SUCCESS if ConnectionID was correctly negotiated, error code otherwise

C.58.4.518 function wolfSSL_dtls_cid_get_rx

```
int wolfSSL_dtls_cid_get_rx(
    WOLFSSL * ssl,
    unsigned char * buffer,
    unsigned int bufferSz
)
```

Copy the ConnectionID used by the other peer to send records in this connection into the buffer pointed by the parameter buffer. See RFC 9146 and RFC.

1. The available space in the buffer need to be provided in bufferSz. WOLFSSL_SUCCESS if ConnectionID was correctly copied, error code otherwise

sslA WOLFSSL object pointern

bufferA buffer where the ConnectionID will be copied

bufferSzavailable space in buffer

wolfSSL_dtls_cid_get0_rx

wolfSSL_dtls_cid_use

wolfSSL_dtls_cid_is_enabled

wolfSSL_dtls_cid_set

wolfSSL_dtls_cid_get_rx_size

wolfSSL_dtls_cid_get_tx_size

wolfSSL_dtls_cid_get_tx

C.58.4.519 function wolfSSL_dtls_cid_get0_rx

```
int wolfSSL_dtls_cid_get0_rx(
    WOLFSSL * ssl,
    unsigned char ** cid
)
```

Get the ConnectionID used by the other peer. See RFC 9146 and RFC 9147.

Parameters:

- **ssl** A WOLFSSL object pointern
- **cid** Pointer that will be set to the internal memory that holds the CID

See:

- wolfSSL_dtls_cid_get_rx
- wolfSSL_dtls_cid_use
- wolfSSL_dtls_cid_is_enabled
- wolfSSL_dtls_cid_set
- wolfSSL_dtls_cid_get_rx_size
- wolfSSL_dtls_cid_get_tx_size
- wolfSSL_dtls_cid_get_tx

Return: WOLFSSL_SUCCESS if ConnectionID was correctly set in cid.

C.58.4.520 function wolfSSL_dtls_cid_get_tx_size

```
int wolfSSL_dtls_cid_get_tx_size(
    WOLFSSL * ssl,
    unsigned int * size
)
```

Get the size of the ConnectionID used to send records in this connection. See RFC 9146 and RFC 9147. The size is stored in the parameter size. The send ConnectionID is chosen by the peer. On DTLS 1.3 it may be up to 255 bytes and is not bounded by wolfSSL_dtls_cid_max_size(); on DTLS 1.2 it never exceeds wolfSSL_dtls_cid_max_size().

Parameters:

- **ssl** A WOLFSSL object pointern
- **size** a pointer to an unsigned int where the size will be stored

See:

- wolfSSL_dtls_cid_use
- wolfSSL_dtls_cid_is_enabled
- wolfSSL_dtls_cid_set
- wolfSSL_dtls_cid_get_rx_size
- wolfSSL_dtls_cid_get_rx
- wolfSSL_dtls_cid_get_tx

Return: WOLFSSL_SUCCESS if ConnectionID size was correctly stored, error code otherwise

C.58.4.521 function wolfSSL_dtls_cid_get_tx

```
int wolfSSL_dtls_cid_get_tx(
    WOLFSSL * ssl,
    unsigned char * buffer,
    unsigned int bufferSz
)
```

Copy the ConnectionID used when sending records in this connection into the buffer pointer by the parameter buffer. See RFC 9146 and RFC 9147. The available size need to be provided in bufferSz. The send ConnectionID is chosen by the peer. On DTLS 1.3 it may be up to 255 bytes and is not bounded by wolfSSL_dtls_cid_max_size(), so size the buffer from wolfSSL_dtls_cid_get_tx_size() (or use 255 bytes) rather than from wolfSSL_dtls_cid_max_size(), otherwise this returns LENGTH_ERROR.

Parameters:

- **ssl** A WOLFSSL object pointern
- **buffer** A buffer where the ConnectionID will be copied
- **bufferSz** available space in buffer

See:

- wolfSSL_dtls_cid_get0_tx
- wolfSSL_dtls_cid_use
- wolfSSL_dtls_cid_is_enabled
- wolfSSL_dtls_cid_set
- wolfSSL_dtls_cid_get_rx_size
- wolfSSL_dtls_cid_get_rx
- wolfSSL_dtls_cid_get_tx_size

Return: WOLFSSL_SUCCESS if ConnectionID was correctly copied, error code otherwise

C.58.4.522 function wolfSSL_dtls_cid_get0_tx

```
int wolfSSL_dtls_cid_get0_tx(
    WOLFSSL * ssl,
    unsigned char ** cid
)
```

Get the ConnectionID used when sending records in this connection. See RFC 9146 and RFC 9147. The send ConnectionID is chosen by the peer. On DTLS 1.3 it may be up to 255 bytes and is not bounded by wolfSSL_dtls_cid_max_size(); use wolfSSL_dtls_cid_get_tx_size() to learn its length.

Parameters:

- **ssl** A WOLFSSL object pointern
- **cid** Pointer that will be set to the internal memory that holds the CID

See:

- wolfSSL_dtls_cid_get_tx
- wolfSSL_dtls_cid_use
- wolfSSL_dtls_cid_is_enabled
- wolfSSL_dtls_cid_set
- wolfSSL_dtls_cid_get_rx_size
- wolfSSL_dtls_cid_get_rx
- wolfSSL_dtls_cid_get_tx_size

Return: WOLFSSL_SUCCESS if ConnectionID was correctly retrieved, error code otherwise

C.58.4.523 function wolfSSL_dtls_cid_max_size

```
int wolfSSL_dtls_cid_max_size(
    void
)
```

Get the maximum size of the ConnectionID that this build can receive, that is the largest size accepted by wolfSSL_dtls_cid_set(). This is the compile time define DTLS_CID_MAX_SIZE and it can never be

bigger than 255 bytes. It does not bound the ConnectionID used to send records, which is chosen by the peer and, on DTLS 1.3, may be up to 255 bytes.

See:

- wolfSSL_dtls_cid_set
- wolfSSL_dtls_cid_get_rx_size
- wolfSSL_dtls_cid_get_tx_size

Return: the maximum receive ConnectionID size in bytes

C.58.4.524 function wolfSSL_dtls_cid_parse

```
const unsigned char * wolfSSL_dtls_cid_parse(
    const unsigned char * msg,
    unsigned int msgSz,
    unsigned int cidSz
)
```

Extract the ConnectionID from a record datagram/message. See RFC 9146 and RFC 9147.

Parameters:

- **msg** buffer holding the datagram read from the network
- **msgSz** size of msg in bytes
- **cid** pointer to the start of the CID inside the msg buffer
- **cidSz** the expected size of the CID. The record layer does not have a CID size field so we have to know beforehand the size of the CID. It is recommended to use a constant CID for all connections.

See:

- wolfSSL_dtls_cid_get_tx
- wolfSSL_dtls_cid_use
- wolfSSL_dtls_cid_is_enabled
- wolfSSL_dtls_cid_set
- wolfSSL_dtls_cid_get_rx_size
- wolfSSL_dtls_cid_get_rx
- wolfSSL_dtls_cid_get_tx_size

C.58.4.525 function wolfSSL_CTX_set_client_CA_list

```
void wolfSSL_CTX_set_client_CA_list(
    WOLFSSL_CTX * ctx,
    WOLF_STACK_OF(WOLFSSL_X509_NAME) * names
)
```

On the server, this sets a list of CA names to be sent to clients in certificate requests as a hint for which CA's are supported by the server.

Parameters:

- **ctx** Pointer to the wolfSSL context
- **names** List of names to be set

See:

- wolfSSL_set_client_CA_list
- wolfSSL_CTX_get_client_CA_list
- wolfSSL_get_client_CA_list
- wolfSSL_CTX_set0_CA_list
- wolfSSL_set0_CA_list

- wolfSSL_CTX_get0_CA_list
- wolfSSL_get0_CA_list
- wolfSSL_get0_peer_CA_list

On the client, this function has no effect.

C.58.4.526 function wolfSSL_CTX_get_client_CA_list

```
WOLFSSL_STACK * wolfSSL_CTX_get_client_CA_list(
    const WOLFSSL_CTX * ctx
)
```

This retrieves the list previously set via wolfSSL_CTX_set_client_CA_list, or NULL if no list has been set.

Parameters:

- **ctx** Pointer to the wolfSSL context

See:

- wolfSSL_set_client_CA_list
- wolfSSL_CTX_set_client_CA_list
- wolfSSL_get_client_CA_list
- wolfSSL_CTX_set0_CA_list
- wolfSSL_set0_CA_list
- wolfSSL_CTX_get0_CA_list
- wolfSSL_get0_CA_list
- wolfSSL_get0_peer_CA_list

Return: A stack of WOLFSSL_X509_NAMEs containing the CA names

C.58.4.527 function wolfSSL_set_client_CA_list

```
void wolfSSL_set_client_CA_list(
    WOLFSSL * ssl,
    WOLF_STACK_OF(WOLFSSL_X509_NAME) * names
)
```

Same as wolfSSL_CTX_set_client_CA_list, but specific to a session. If a CA list is set on both the context and the session, the list on the session is used.

Parameters:

- **ssl** Pointer to the WOLFSSL object
- **names** List of names to be set.

See:

- wolfSSL_CTX_set_client_CA_list
- wolfSSL_CTX_get_client_CA_list
- wolfSSL_get_client_CA_list
- wolfSSL_CTX_set0_CA_list
- wolfSSL_set0_CA_list
- wolfSSL_CTX_get0_CA_list
- wolfSSL_get0_CA_list
- wolfSSL_get0_peer_CA_list

C.58.4.528 function wolfSSL_get_client_CA_list

```
WOLFSSL_STACK * wolfSSL_get_client_CA_list(
    const WOLFSSL * ssl
)
```

On the server, this retrieves the list previously set via wolfSSL_set_client_CA_list. If none was set, returns the list previously set via wolfSSL_CTX_set_client_CA_list. If no list at all was set, returns NULL.

Parameters:

- **ssl** Pointer to the WOLFSSL object

See:

- wolfSSL_CTX_set_cert_cb
- wolfSSL_CTX_set_client_CA_list
- wolfSSL_CTX_get_client_CA_list
- wolfSSL_get_client_CA_list
- wolfSSL_CTX_set0_CA_list
- wolfSSL_set0_CA_list
- wolfSSL_CTX_get0_CA_list
- wolfSSL_get0_CA_list
- wolfSSL_get0_peer_CA_list

Return: A stack of WOLFSSL_X509_NAMEs containing the CA names

On the client, this retrieves the list that was received from the server, or NULL if none was received. wolfSSL_CTX_set_cert_cb can be used to register a callback to dynamically load certificates when a certificate request is received from the server.

C.58.4.529 function wolfSSL_CTX_set0_CA_list

```
void wolfSSL_CTX_set0_CA_list(
    WOLFSSL_CTX * ctx,
    WOLF_STACK_OF(WOLFSSL_X509_NAME) * names
)
```

This function sets a list of CA names to be sent to the peer as a hint for which CA's are supported for its authentication.

Parameters:

- **ctx** Pointer to the wolfSSL context
- **names** List of names to be set

See:

- wolfSSL_CTX_set_client_CA_list
- wolfSSL_set_client_CA_list
- wolfSSL_CTX_get_client_CA_list
- wolfSSL_get_client_CA_list
- wolfSSL_set0_CA_list
- wolfSSL_CTX_get0_CA_list
- wolfSSL_get0_CA_list
- wolfSSL_get0_peer_CA_list

In TLS >= 1.3, this is supported in both directions between the client and the server. On the server, the CA names will be sent as part of a CertificateRequest, making this function an equivalent of *_set_client_CA_list; on the client, these are sent as part of ClientHello.

In TLS < 1.3, sending CA names from the client to the server is not supported, therefore this function is equivalent to wolfSSL_CTX_set_client_CA_list.

Note that the lists set via *_set_client_CA_list and *_set0_CA_list are separate internally, i.e. calling *_get_client_CA_list will not retrieve a list set via *_set0_CA_list and vice versa. If both are set, the server will ignore *_set0_CA_list when sending CA names to the client.

C.58.4.530 function wolfSSL_CTX_get0_CA_list

```
WOLFSSL_STACK * wolfSSL_CTX_get0_CA_list(
    const WOLFSSL_CTX * ctx
)
```

This retrieves the list previously set via wolfSSL_CTX_set0_CA_list, or NULL if no list has been set.

Parameters:

- **ctx** Pointer to the wolfSSL context

See:

- wolfSSL_CTX_set_client_CA_list
- wolfSSL_set_client_CA_list
- wolfSSL_CTX_get_client_CA_list
- wolfSSL_get_client_CA_list
- wolfSSL_CTX_set0_CA_list
- wolfSSL_set0_CA_list
- wolfSSL_get0_CA_list
- wolfSSL_get0_peer_CA_list

Return: A stack of WOLFSSL_X509_NAMEs containing the CA names

C.58.4.531 function wolfSSL_set0_CA_list

```
void wolfSSL_set0_CA_list(
    WOLFSSL * ssl,
    WOLF_STACK_OF(WOLFSSL_X509_NAME) * names
)
```

Same as wolfSSL_CTX_set0_CA_list, but specific to a session. If a CA list is set on both the context and the session, the list on the session is used.

Parameters:

- **ssl** Pointer to the WOLFSSL object
- **names** List of names to be set.

See:

- wolfSSL_CTX_set_client_CA_list
- wolfSSL_set_client_CA_list
- wolfSSL_CTX_get_client_CA_list
- wolfSSL_get_client_CA_list
- wolfSSL_CTX_set0_CA_list
- wolfSSL_CTX_get0_CA_list
- wolfSSL_get0_CA_list
- wolfSSL_get0_peer_CA_list

C.58.4.532 function wolfSSL_get0_CA_list

```
WOLFSSL_STACK * wolfSSL_get0_CA_list(
    const WOLFSSL * ssl
)
```

This retrieves the list previously set via wolfSSL_set0_CA_list. If none was set, returns the list previously set via wolfSSL_CTX_set0_CA_list. If no list at all was set, returns NULL.

Parameters:

- **ssl** Pointer to the WOLFSSL object

See:

- wolfSSL_CTX_set_client_CA_list
- wolfSSL_set_client_CA_list
- wolfSSL_CTX_get_client_CA_list
- wolfSSL_get_client_CA_list
- wolfSSL_CTX_set0_CA_list
- wolfSSL_set0_CA_list
- wolfSSL_CTX_get0_CA_list
- wolfSSL_get0_peer_CA_list

Return: A stack of WOLFSSL_X509_NAMEs containing the CA names

C.58.4.533 function wolfSSL_get0_peer_CA_list

```
WOLFSSL_STACK * wolfSSL_get0_peer_CA_list(
    const WOLFSSL * ssl
)
```

This returns the CA list received from the peer.

Parameters:

- **ssl** Pointer to the WOLFSSL object

See:

- wolfSSL_CTX_set_cert_cb
- wolfSSL_CTX_set_client_CA_list
- wolfSSL_set_client_CA_list
- wolfSSL_CTX_get_client_CA_list
- wolfSSL_get_client_CA_list
- wolfSSL_CTX_set0_CA_list
- wolfSSL_set0_CA_list
- wolfSSL_CTX_get0_CA_list
- wolfSSL_get0_CA_list

Return: A stack of WOLFSSL_X509_NAMEs containing the CA names

On the client, this is the list sent by the server in a CertificateRequest, and this function is equivalent to wolfSSL_get_client_CA_list.

On the server, this is the list sent by the client in the ClientHello message in TLS >= 1.3; in TLS < 1.3, the function always returns NULL on the server side.

wolfSSL_CTX_set_cert_cb can be used to register a callback to dynamically load certificates when a CA list is received from the peer.

C.58.4.534 function wolfSSL_get_client_suites_sigalgs

```
int wolfSSL_get_client_suites_sigalgs(
    const WOLFSSL * ssl,
    const byte ** suites,
    word16 * suiteSz,
    const byte ** hashSigAlgo,
    word16 * hashSigAlgoSz
)
```

This function returns the raw list of ciphersuites and signature algorithms offered by the client. The lists are only stored and returned inside a callback setup with wolfSSL_CTX_set_cert_cb(). This is useful to be able to dynamically load certificates and keys based on the available ciphersuites and signature algorithms.

Parameters:

- **ssl** The WOLFSSL object to extract the lists from.
- **suites** Raw and unfiltered list of client ciphersuites. May be NULL if no suites are available.
- **suiteSz** Size of suites in bytes.
- **hashSigAlgo** Raw and unfiltered list of client signature algorithms. May be NULL if not provided.
- **hashSigAlgoSz** Size of hashSigAlgo in bytes.

See:

- wolfSSL_get_ciphersuite_info
- wolfSSL_get_sigalg_info

Return:

- WOLFSSL_SUCCESS when suites available
- WOLFSSL_FAILURE when suites not available

Example

```
int certCB(WOLFSSL* ssl, void* arg)
{
    const byte* suites = NULL;
    word16 suiteSz = 0;
    const byte* hashSigAlgo = NULL;
    word16 hashSigAlgoSz = 0;

    wolfSSL_get_client_suites_sigalgs(ssl, &suites, &suiteSz, &hashSigAlgo,
            &hashSigAlgoSz);

    // Choose certificate to load based on ciphersuites and sigalgs
}

WOLFSSL* ctx;
ctx  = wolfSSL_CTX_new(wolfTLSv1_3_method_ex(NULL));
wolfSSL_CTX_set_cert_cb(ctx, certCB, NULL);
```

C.58.4.535 function wolfSSL_get_ciphersuite_info

```
WOLFSSL_CIPHERSUITE_INFO wolfSSL_get_ciphersuite_info(
    byte first,
    byte second
)
```

This returns information about the ciphersuite directly from the raw ciphersuite bytes.

Parameters:

- **first** First byte of the ciphersuite
- **second** Second byte of the ciphersuite

See:

- wolfSSL_get_client_suites_sigalgs
- wolfSSL_get_sigalg_info

Return: WOLFSSL_CIPHERSUITE_INFO A struct containing information about the type of authentication used in the ciphersuite.

Example

```
WOLFSSL_CIPHERSUITE_INFO info =
        wolfSSL_get_ciphersuite_info(suites[0], suites[1]);
if (info.rsaAuth)
    haveRSA = 1;
else if (info.eccAuth)
    haveECC = 1;
```

C.58.4.536 function wolfSSL_get_sigalg_info

```
int wolfSSL_get_sigalg_info(
    byte first,
    byte second,
    int * hashAlgo,
    int * sigAlgo
)
```

This returns information about the hash and signature algorithm directly from the raw ciphersuite bytes.

Parameters:

- **first** First byte of the hash and signature algorithm
- **second** Second byte of the hash and signature algorithm
- **hashAlgo** The enum wc_HashType of the MAC algorithm
- **sigAlgo** The enum Key_Sum of the authentication algorithm

See:

- wolfSSL_get_client_suites_sigalgs
- wolfSSL_get_ciphersuite_info

Return:

- 0 when info was correctly set
- BAD_FUNC_ARG when either input parameters are NULL or the bytes are not a recognized sigalg suite

Example

```
enum wc_HashType hashAlgo;
enum Key_Sum sigAlgo;

wolfSSL_get_sigalg_info(hashSigAlgo[idx+0], hashSigAlgo[idx+1],
        &hashAlgo, &sigAlgo);
```

```
if (sigAlgo == RSAk || sigAlgo == RSAPSSk)
    haveRSA = 1;
else if (sigAlgo == ECDSAk)
    haveECC = 1;
```

C.58.4.537 function wolfSSL_CTX_set_default_passwd_cb

```
void wolfSSL_CTX_set_default_passwd_cb(
    WOLFSSL_CTX * ctx,
    wc_pem_password_cb * cb
)
```

This function will set the password callback in the provided CTX. This callback is used when loading an encrypted cert or key which requires a password.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().
- **cb** a function pointer to (*wc_pem_password_cb) that is set to the passwd_cb member of the WOLFSSL_CTX.

See: wolfSSL_CTX_set_default_passwd_cb_userdata

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
int PasswordCallBack(char* passwd, int sz, int rw, void* userdata) {

}
…
wolfSSL_CTX_set_default_passwd_cb(ctx, PasswordCallBack);
```

C.58.4.538 function wolfSSL_CTX_set_default_passwd_cb_userdata

```
void wolfSSL_CTX_set_default_passwd_cb_userdata(
    WOLFSSL_CTX * ctx,
    void * userdata
)
```

This function will set the userdata argument to the passwd_userdata member of the WOLFSSL_CTX structure. This member is passed into the CTX's password callback when called.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().
- **userdata** a pointer to userdata which is passed into the password callback.

See: wolfSSL_CTX_set_default_passwd_cb

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
int data;
…
wolfSSL_CTX_set_default_passwd_cb_userdata(ctx, (void*)&data);
```

C.58.4.539 function wolfSSL_get_scr_check_enabled

```
int wolfSSL_get_scr_check_enabled(
    const WOLFSSL * ssl
)
```

Gets the state of the secure renegotiation (SCR) check requirement.

Parameters:

- **ssl** Pointer to the WOLFSSL structure, created with wolfSSL_new().

See: wolfSSL_set_scr_check_enabled

Return:

- 1 if the SCR check is enabled.
- 0 if the SCR check is disabled.
- BAD_FUNC_ARG if ssl is NULL.

This function returns whether the client requires the server to acknowledge the secure renegotiation extension and enable secure renegotiation when sending it from the client. When enabled, the client will generate a fatal handshake_failure alert if the server does not acknowledge the extension in the ServerHello message, as required by RFC 9325.

Example

```
WOLFSSL* ssl;
int enabled;

ssl = wolfSSL_new(ctx);
enabled = wolfSSL_get_scr_check_enabled(ssl);
if (enabled) {
    // SCR check is enabled
}
```

C.58.4.540 function wolfSSL_set_scr_check_enabled

```
int wolfSSL_set_scr_check_enabled(
    WOLFSSL * ssl,
    byte enabled
)
```

Sets the state of the secure renegotiation (SCR) check requirement.

Parameters:

- **ssl** Pointer to the WOLFSSL structure, created with wolfSSL_new().
- **enabled** Non-zero to enable the SCR check, zero to disable it.

See: wolfSSL_get_scr_check_enabled

Return:

- WOLFSSL_SUCCESS on success.
- BAD_FUNC_ARG if ssl is NULL.

This function enables or disables the requirement for the server to acknowledge the secure renegotiation extension and enable secure renegotiation when sending it from the client. When enabled, the client will generate a fatal handshake_failure alert if the server does not acknowledge the extension in the ServerHello message, as required by RFC 9325.

Example

```
WOLFSSL* ssl;
int ret;

ssl = wolfSSL_new(ctx);
ret = wolfSSL_set_scr_check_enabled(ssl, 1);
if (ret != WOLFSSL_SUCCESS) {
    // Error setting SCR check
}
```

C.58.4.541 function wolfSSL_CTX_DisableExtendedMasterSecret

```
int wolfSSL_CTX_DisableExtendedMasterSecret(
    WOLFSSL_CTX * ctx
)
```

Disables the TLS Extended Master Secret extension (RFC 7627) on the context: a client stops advertising it and a server ignores the peer's request, so a standard master secret is negotiated. A server also declines resumption of sessions or tickets that used EMS and does a full handshake instead. TLS 1.2 and earlier only. Requires HAVE_EXTENDED_MASTER.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

See:

- wolfSSL_DisableExtendedMasterSecret
- wolfSSL_CTX_EnableExtendedMasterSecret
- wolfSSL_CTX_RequireExtendedMasterSecret

Return:

- WOLFSSL_SUCCESS on success.
- BAD_FUNC_ARG if ctx is NULL.

Example

```
wolfSSL_CTX_DisableExtendedMasterSecret(ctx);
```

C.58.4.542 function wolfSSL_DisableExtendedMasterSecret

```
int wolfSSL_DisableExtendedMasterSecret(
    WOLFSSL * ssl
)
```

Disables the TLS Extended Master Secret extension (RFC 7627) on the SSL object: a client stops advertising it and a server ignores the peer's request, so a standard master secret is negotiated. A server also declines resumption of sessions or tickets that used EMS and does a full handshake instead. Call before the handshake starts, or after wolfSSL_clear. TLS 1.2 and earlier only. Requires HAVE_EXTENDED_MASTER.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_CTX_DisableExtendedMasterSecret
- wolfSSL_EnableExtendedMasterSecret
- wolfSSL_RequireExtendedMasterSecret

Return:

- WOLFSSL_SUCCESS on success.
- BAD_FUNC_ARG if ssl is NULL.
- BAD_STATE_E if the handshake has started.

Example

```
wolfSSL_DisableExtendedMasterSecret(ssl);
```

C.58.4.543 function wolfSSL_CTX_EnableExtendedMasterSecret

```
int wolfSSL_CTX_EnableExtendedMasterSecret(
    WOLFSSL_CTX * ctx
)
```

Re-enables the TLS Extended Master Secret extension (RFC 7627) on the context (the default): EMS is used when the peer supports it but is not mandatory. Undoes a previous disable or require. Requires HAVE_EXTENDED_MASTER.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

See:

- wolfSSL_EnableExtendedMasterSecret
- wolfSSL_CTX_DisableExtendedMasterSecret
- wolfSSL_CTX_RequireExtendedMasterSecret

Return:

- WOLFSSL_SUCCESS on success.
- BAD_FUNC_ARG if ctx is NULL.

Example

```
wolfSSL_CTX_EnableExtendedMasterSecret(ctx);
```

C.58.4.544 function wolfSSL_EnableExtendedMasterSecret

```
int wolfSSL_EnableExtendedMasterSecret(
    WOLFSSL * ssl
)
```

Re-enables the TLS Extended Master Secret extension (RFC 7627) on the SSL object (the default): EMS is used when the peer supports it but is not mandatory. Undoes a previous disable or require. Call before the handshake starts, or after wolfSSL_clear. Requires HAVE_EXTENDED_MASTER.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_CTX_EnableExtendedMasterSecret
- wolfSSL_DisableExtendedMasterSecret
- wolfSSL_RequireExtendedMasterSecret

Return:

- WOLFSSL_SUCCESS on success.
- BAD_FUNC_ARG if ssl is NULL.
- BAD_STATE_E if the handshake has started.

Example

```
wolfSSL_EnableExtendedMasterSecret(ssl);
```

C.58.4.545 function wolfSSL_CTX_RequireExtendedMasterSecret

```
int wolfSSL_CTX_RequireExtendedMasterSecret(
    WOLFSSL_CTX * ctx
)
```

Makes the TLS Extended Master Secret extension (RFC 7627) mandatory on the context: if it is not negotiated, the connection is aborted with EXT_MASTER_SECRET_NEEDED_E. A client advertises the extension even after a previous disable. TLS 1.2 and earlier only. Requires HAVE_EXTENDED_MASTER.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

See:

- wolfSSL_RequireExtendedMasterSecret
- wolfSSL_CTX_EnableExtendedMasterSecret
- wolfSSL_CTX_DisableExtendedMasterSecret

Return:

- WOLFSSL_SUCCESS on success.
- BAD_FUNC_ARG if ctx is NULL.

Example

```
wolfSSL_CTX_RequireExtendedMasterSecret(ctx);
```

C.58.4.546 function wolfSSL_RequireExtendedMasterSecret

```
int wolfSSL_RequireExtendedMasterSecret(
    WOLFSSL * ssl
)
```

Makes the TLS Extended Master Secret extension (RFC 7627) mandatory on the SSL object: if it is not negotiated, including on resumption, the connection is aborted with EXT_MASTER_SECRET_NEEDED_E. A client advertises the extension even after a previous disable. Call before the handshake starts, or after wolfSSL_clear. TLS 1.2 and earlier only. Requires HAVE_EXTENDED_MASTER.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_CTX_RequireExtendedMasterSecret
- wolfSSL_EnableExtendedMasterSecret
- wolfSSL_DisableExtendedMasterSecret

Return:

- WOLFSSL_SUCCESS on success.
- BAD_FUNC_ARG if ssl is NULL.
- BAD_STATE_E if the handshake has started.

Example

```
wolfSSL_RequireExtendedMasterSecret(ssl);
```

C.58.4.547 function wolfSSL_UseCertificateAuthority

```
int wolfSSL_UseCertificateAuthority(
    WOLFSSL * ssl,
    const unsigned char * dn,
    unsigned int dnSz
)
```

Adds a CA distinguished name to the list of certificate authorities announced via the TLS 1.3 certificate_authorities extension (RFC 8446 section 4.2.4) on the given SSL session. The DN must be the inner content of a DER-encoded X.509 Name (the bytes after the SEQUENCE tag and length), as returned by wc_GetDecodedCertSubjectRaw(). The library copies the bytes and prepends the SEQUENCE header on the wire automatically.

Parameters:

- **ssl** pointer to a WOLFSSL object, created with wolfSSL_new().
- **dn** pointer to the DER-encoded subject Name content.
- **dnSz** size in bytes of the DN content.

See:

- wolfSSL_CTX_UseCertificateAuthority
- wolfSSL_ClearCertificateAuthorities
- wolfSSL_GetPeerCertificateAuthorityCount
- wolfSSL_GetPeerCertificateAuthority
- wc_GetDecodedCertSubjectRaw

Return:

- 0 on success.
- BAD_FUNC_ARG if ssl or dn is NULL, dnSz is 0, or dnSz exceeds the maximum content size.
- MEMORY_ERROR if memory allocation fails.

Multiple DNs may be added; each call appends to the list. Use wolfSSL_ClearCertificateAuthorities() to reset the list.

Requires WOLFSSL_TLS13 and !NO_CERTS and !WOLFSSL_NO_CA_NAMES.

Example

```
DecodedCert decoded;
const byte* subject = NULL;
int subjectSz = 0;

wc_InitDecodedCert(&decoded, certDer, certDerSz, NULL);
wc_ParseCert(&decoded, CERT_TYPE, NO_VERIFY, NULL);
wc_GetDecodedCertSubjectRaw(&decoded, &subject, &subjectSz);

ret = wolfSSL_UseCertificateAuthority(ssl, subject,
        (unsigned int)subjectSz);
if (ret != 0) {
    // error adding CA DN
}
wc_FreeDecodedCert(&decoded);
```

C.58.4.548 function wolfSSL_CTX_UseCertificateAuthority

```
int wolfSSL_CTX_UseCertificateAuthority(
    WOLFSSL_CTX * ctx,
```

```
    const unsigned char * dn,
    unsigned int dnSz
)
```

Adds a CA distinguished name to the list of certificate authorities announced via the TLS 1.3 certificate_authorities extension (RFC 8446 section 4.2.4) on all SSL sessions created from this context. The DN format and requirements are identical to wolfSSL_UseCertificateAuthority().

Parameters:

- **ctx** pointer to a WOLFSSL_CTX object, created with wolfSSL_CTX_new().
- **dn** pointer to the DER-encoded subject Name content.
- **dnSz** size in bytes of the DN content.

See:

- wolfSSL_UseCertificateAuthority
- wolfSSL_CTX_ClearCertificateAuthorities
- wc_GetDecodedCertSubjectRaw

Return:

- 0 on success.
- BAD_FUNC_ARG if ctx or dn is NULL, dnSz is 0, or dnSz exceeds the maximum content size.
- MEMORY_ERROR if memory allocation fails.

Per-session lists set via wolfSSL_UseCertificateAuthority() take precedence; if the SSL object has its own list, the CTX list is not sent.

Requires WOLFSSL_TLS13 and !NO_CERTS and !WOLFSSL_NO_CA_NAMES.

Example

```
ret = wolfSSL_CTX_UseCertificateAuthority(ctx, subject,
        (unsigned int)subjectSz);
if (ret != 0) {
    // error adding CA DN
}
```

C.58.4.549 function wolfSSL_ClearCertificateAuthorities

```
void wolfSSL_ClearCertificateAuthorities(
    WOLFSSL * ssl
)
```

Frees and removes all CA distinguished names previously added to the SSL session via wolfSSL_UseCertificateAuthority(). After this call the session_level native CA list is empty; the CTX_level list (if any) is not affected.

Parameters:

- **ssl** pointer to a WOLFSSL object, created with wolfSSL_new().

See:

- wolfSSL_UseCertificateAuthority
- wolfSSL_CTX_ClearCertificateAuthorities

Return: none No return value.

Requires WOLFSSL_TLS13 and !NO_CERTS and !WOLFSSL_NO_CA_NAMES.

Example

```
wolfSSL_UseCertificateAuthority(ssl, dn1, dn1Sz);
wolfSSL_UseCertificateAuthority(ssl, dn2, dn2Sz);
// Clear all session-level CA DNs:
wolfSSL_ClearCertificateAuthorities(ssl);
```

C.58.4.550 function wolfSSL_CTX_ClearCertificateAuthorities

```
void wolfSSL_CTX_ClearCertificateAuthorities(
    WOLFSSL_CTX * ctx
)
```

Frees and removes all CA distinguished names previously added to the context via wolfSSL_CTX_UseCertificateAuthorit
After this call the CTX-level native CA list is empty.

Parameters:

- **ctx** pointer to a WOLFSSL_CTX object, created with wolfSSL_CTX_new().

See:

- wolfSSL_CTX_UseCertificateAuthority
- wolfSSL_ClearCertificateAuthorities

Return: none No return value.

Requires WOLFSSL_TLS13 and !NO_CERTS and !WOLFSSL_NO_CA_NAMES.

Example

```
wolfSSL_CTX_UseCertificateAuthority(ctx, dn, dnSz);
// Clear all CTX-level CA DNs:
wolfSSL_CTX_ClearCertificateAuthorities(ctx);
```

C.58.4.551 function wolfSSL_GetPeerCertificateAuthorityCount

```
int wolfSSL_GetPeerCertificateAuthorityCount(
    const WOLFSSL * ssl
)
```

Returns the number of CA distinguished names received from the peer in the TLS 1.3 certificate_authorities extension. This is typically called inside a cert_cb (WOLFSSL_CERT_SETUP_CB) on the server side to inspect which CAs the client trusts.

Parameters:

- **ssl** pointer to a WOLFSSL object, created with wolfSSL_new().

See:

- wolfSSL_GetPeerCertificateAuthority
- wolfSSL_UseCertificateAuthority

Return: >= 0 The number of peer CA DNs. Returns 0 if ssl is NULL or no certificate_authorities extension was received.

Requires WOLFSSL_TLS13 and !NO_CERTS and !WOLFSSL_NO_CA_NAMES.

Example

```
int count = wolfSSL_GetPeerCertificateAuthorityCount(ssl);
for (int i = 0; i < count; i++) {
    int sz = wolfSSL_GetPeerCertificateAuthority(ssl, i, NULL, 0);
```

```
    // sz is the DN size in bytes
}
```

C.58.4.552 function wolfSSL_GetPeerCertificateAuthority

```
int wolfSSL_GetPeerCertificateAuthority(
    const WOLFSSL * ssl,
    int idx,
    unsigned char * outDn,
    unsigned int outDnSz
)
```

Copies the idx-th CA distinguished name received from the peer in the TLS 1.3 certificate_authorities extension into the caller's buffer. The DN is the inner content of the DER-encoded Name (without the SEQUENCE header), matching the format accepted by wolfSSL_UseCertificateAuthority().

Parameters:

- **ssl** pointer to a WOLFSSL object, created with wolfSSL_new().
- **idx** zero-based index of the peer CA DN to retrieve. Must be less than the count returned by wolfSSL_GetPeerCertificateAuthorityCount().
- **outDn** output buffer to receive the DN bytes, or NULL to query size.
- **outDnSz** size of the output buffer in bytes.

See:

- wolfSSL_GetPeerCertificateAuthorityCount
- wolfSSL_UseCertificateAuthority
- wc_GetDecodedCertSubjectRaw

Return:

- 0 The number of bytes written to outDn, or the DN size if outDn is NULL.
- BAD_FUNC_ARG if ssl is NULL or idx is out of range.
- BUFFER_E if outDnSz is smaller than the DN.

If outDn is NULL, returns the size of the DN in bytes (allowing the caller to allocate the right amount of memory). If outDn is non-NULL and outDnSz is large enough, copies the DN bytes and returns the number of bytes written. If outDnSz is too small, returns BUFFER_E.

Requires WOLFSSL_TLS13 and !NO_CERTS and !WOLFSSL_NO_CA_NAMES.

Example

```
int count = wolfSSL_GetPeerCertificateAuthorityCount(ssl);
for (int i = 0; i < count; i++) {
    int sz = wolfSSL_GetPeerCertificateAuthority(ssl, i, NULL, 0);
    if (sz > 0) {
        unsigned char* dn = malloc(sz);
        wolfSSL_GetPeerCertificateAuthority(ssl, i, dn,
                (unsigned int)sz);
        // use dn[0..sz-1]
        free(dn);
    }
}
```

C.58.5 Source code

```
WOLFSSL_METHOD *wolfDTLSv1_2_client_method_ex(void* heap);

WOLFSSL_METHOD *wolfSSLv23_method(void);

WOLFSSL_METHOD *wolfSSLv3_server_method(void);

WOLFSSL_METHOD *wolfSSLv3_client_method(void);

WOLFSSL_METHOD *wolfTLSv1_server_method(void);

WOLFSSL_METHOD *wolfTLSv1_client_method(void);

WOLFSSL_METHOD *wolfTLSv1_1_server_method(void);

WOLFSSL_METHOD *wolfTLSv1_1_client_method(void);

WOLFSSL_METHOD *wolfTLSv1_2_server_method(void);

WOLFSSL_METHOD *wolfTLSv1_2_client_method(void);

WOLFSSL_METHOD *wolfDTLSv1_client_method(void);

WOLFSSL_METHOD *wolfDTLSv1_server_method(void);
WOLFSSL_METHOD *wolfDTLSv1_3_server_method(void);
WOLFSSL_METHOD* wolfDTLSv1_3_client_method(void);
WOLFSSL_METHOD *wolfDTLS_server_method(void);
WOLFSSL_METHOD *wolfDTLS_client_method(void);
WOLFSSL_METHOD *wolfDTLSv1_2_server_method(void);

int wolfSSL_use_old_poly(WOLFSSL* ssl, int value);

int wolfSSL_dtls_import(WOLFSSL* ssl, const unsigned char* buf,
                                                        unsigned int sz);


int wolfSSL_tls_import(WOLFSSL* ssl, const unsigned char* buf,
        unsigned int sz);

int wolfSSL_CTX_dtls_set_export(WOLFSSL_CTX* ctx,
                                                    wc_dtls_export func);

int wolfSSL_dtls_set_export(WOLFSSL* ssl, wc_dtls_export func);

int wolfSSL_dtls_export(WOLFSSL* ssl, unsigned char* buf,
                                                      unsigned int* sz);

int wolfSSL_tls_export(WOLFSSL* ssl, unsigned char* buf,
        unsigned int* sz);

int wolfSSL_CTX_load_static_memory(WOLFSSL_CTX** ctx,
                                           wolfSSL_method_func method,
```

```
                                            unsigned char* buf, unsigned int sz,
                                             int flag, int max);

int wolfSSL_CTX_is_static_memory(WOLFSSL_CTX* ctx,
                                                 WOLFSSL_MEM_STATS* mem_stats);

int wolfSSL_is_static_memory(WOLFSSL* ssl,
                                              WOLFSSL_MEM_CONN_STATS* mem_stats);

int wolfSSL_CTX_use_certificate_file(WOLFSSL_CTX* ctx, const char* file,
                                     int format);

int wolfSSL_CTX_use_PrivateKey_file(WOLFSSL_CTX* ctx, const char* file, int
 ↪  format);

int wolfSSL_CTX_load_verify_locations(WOLFSSL_CTX* ctx, const char* file,
                                                 const char* path);

int wolfSSL_CTX_load_verify_locations_ex(WOLFSSL_CTX* ctx, const char* file,
                                         const char* path, word32 flags);

const char** wolfSSL_get_system_CA_dirs(word32* num);

int wolfSSL_CTX_load_system_CA_certs(WOLFSSL_CTX* ctx);

int wolfSSL_CTX_trust_peer_cert(WOLFSSL_CTX* ctx, const char* file, int type);

int wolfSSL_CTX_use_certificate_chain_file(WOLFSSL_CTX *ctx,
                                                       const char *file);

int wolfSSL_CTX_use_RSAPrivateKey_file(WOLFSSL_CTX* ctx, const char* file, int
 ↪  format);

long wolfSSL_get_verify_depth(WOLFSSL* ssl);

long wolfSSL_CTX_get_verify_depth(WOLFSSL_CTX* ctx);

int wolfSSL_use_certificate_file(WOLFSSL* ssl, const char* file, int format);

int wolfSSL_use_PrivateKey_file(WOLFSSL* ssl, const char* file, int format);

int wolfSSL_use_certificate_chain_file(WOLFSSL* ssl, const char *file);

int wolfSSL_use_RSAPrivateKey_file(WOLFSSL* ssl, const char* file, int format);

int wolfSSL_CTX_der_load_verify_locations(WOLFSSL_CTX* ctx,
                                         const char* file, int format);

WOLFSSL_CTX* wolfSSL_CTX_new(WOLFSSL_METHOD*);

WOLFSSL* wolfSSL_new(WOLFSSL_CTX*);

int  wolfSSL_set_fd(WOLFSSL* ssl, int fd);
```

```
int wolfSSL_set_dtls_fd_connected(WOLFSSL* ssl, int fd);
int wolfDTLS_SetChGoodCb(WOLFSSL* ssl, ClientHelloGoodCb cb, void* user_ctx);
char* wolfSSL_get_cipher_list(int priority);
int  wolfSSL_get_ciphers(char* buf, int len);
const char* wolfSSL_get_cipher_name(WOLFSSL* ssl);
int  wolfSSL_get_fd(const WOLFSSL* ssl);
int  wolfSSL_get_wfd(const WOLFSSL* ssl);
void wolfSSL_set_using_nonblock(WOLFSSL* ssl, int nonblock);
int  wolfSSL_get_using_nonblock(WOLFSSL*);
int  wolfSSL_write(WOLFSSL* ssl, const void* data, int sz);
int wolfSSL_write_ex(WOLFSSL* ssl, const void* data, size_t sz, size_t* wr);
int  wolfSSL_read(WOLFSSL* ssl, void* data, int sz);
int wolfSSL_read_ex(WOLFSSL* ssl, void* data, size_t sz, size_t* rd);
int  wolfSSL_peek(WOLFSSL* ssl, void* data, int sz);
int  wolfSSL_accept(WOLFSSL* ssl);
int  wolfDTLS_accept_stateless(WOLFSSL* ssl);
int wolfSSL_disable_cookie(WOLFSSL* ssl);
int wolfSSL_enable_cookie(WOLFSSL* ssl);
void wolfSSL_CTX_free(WOLFSSL_CTX* ctx);
void wolfSSL_free(WOLFSSL* ssl);
int  wolfSSL_shutdown(WOLFSSL* ssl);
int wolfSSL_SendUserCanceled(WOLFSSL* ssl);
int  wolfSSL_send(WOLFSSL* ssl, const void* data, int sz, int flags);
int  wolfSSL_recv(WOLFSSL* ssl, void* data, int sz, int flags);
int  wolfSSL_get_error(WOLFSSL* ssl, int ret);
int  wolfSSL_get_alert_history(WOLFSSL* ssl, WOLFSSL_ALERT_HISTORY *h);
int        wolfSSL_set_session(WOLFSSL* ssl, WOLFSSL_SESSION* session);
```

```
WOLFSSL_SESSION* wolfSSL_get_session(WOLFSSL* ssl);

void       wolfSSL_flush_sessions(WOLFSSL_CTX* ctx, long tm);

int        wolfSSL_SetServerID(WOLFSSL* ssl, const unsigned char* id,
                                       int len, int newSession);

int wolfSSL_GetSessionIndex(WOLFSSL* ssl);

int wolfSSL_GetSessionAtIndex(int index, WOLFSSL_SESSION* session);

    WOLFSSL_X509_CHAIN* wolfSSL_SESSION_get_peer_chain(WOLFSSL_SESSION*
 ↪  session);

void wolfSSL_CTX_set_verify(WOLFSSL_CTX* ctx, int mode,
                                     VerifyCallback verify_callback);

void wolfSSL_set_verify(WOLFSSL* ssl, int mode, VerifyCallback
 ↪  verify_callback);

void wolfSSL_SetCertCbCtx(WOLFSSL* ssl, void* ctx);

void wolfSSL_CTX_SetCertCbCtx(WOLFSSL_CTX* ctx, void* userCtx);

int wolfSSL_CTX_SetChainVerifyCb(WOLFSSL_CTX* ctx, ChainVerifyCb cb);

int wolfSSL_SetChainVerifyCb(WOLFSSL* ssl, ChainVerifyCb cb);

void wolfSSL_CTX_SetChainVerifyCtx(WOLFSSL_CTX* ctx, void* userCtx);

void wolfSSL_SetChainVerifyCtx(WOLFSSL* ssl, void* ctx);

void* wolfSSL_GetChainVerifyCtx(const WOLFSSL* ssl);

int  wolfSSL_pending(WOLFSSL* ssl);

void wolfSSL_load_error_strings(void);

int  wolfSSL_library_init(void);

int wolfSSL_SetDevId(WOLFSSL* ssl, int devId);

int wolfSSL_CTX_SetDevId(WOLFSSL_CTX* ctx, int devId);

int wolfSSL_CTX_GetDevId(WOLFSSL_CTX* ctx, WOLFSSL* ssl);

long wolfSSL_CTX_set_session_cache_mode(WOLFSSL_CTX* ctx, long mode);

void wolfSSL_CTX_sess_set_get_cb(WOLFSSL_CTX* ctx,
    WOLFSSL_SESSION*(*f)(WOLFSSL* ssl, const unsigned char*, int, int*));

int  wolfSSL_set_session_secret_cb(WOLFSSL* ssl, SessionSecretCb cb, void*
 ↪  ctx);
```

```
int  wolfSSL_save_session_cache(const char* fname);

int  wolfSSL_restore_session_cache(const char* fname);

int  wolfSSL_memsave_session_cache(void* mem, int sz);

int  wolfSSL_memrestore_session_cache(const void* mem, int sz);

int  wolfSSL_get_session_cache_memsize(void);

int  wolfSSL_CTX_save_cert_cache(WOLFSSL_CTX* ctx, const char* fname);

int  wolfSSL_CTX_restore_cert_cache(WOLFSSL_CTX* ctx, const char* fname);

int  wolfSSL_CTX_memsave_cert_cache(WOLFSSL_CTX* ctx, void* mem, int sz, int*
 ↪  used);

int  wolfSSL_CTX_memrestore_cert_cache(WOLFSSL_CTX* ctx, const void* mem, int
 ↪  sz);

int  wolfSSL_CTX_get_cert_cache_memsize(WOLFSSL_CTX* ctx);

int  wolfSSL_CTX_set_cipher_list(WOLFSSL_CTX* ctx, const char* list);

int  wolfSSL_set_cipher_list(WOLFSSL* ssl, const char* list);

void wolfSSL_dtls_set_using_nonblock(WOLFSSL* ssl, int nonblock);
int  wolfSSL_dtls_get_using_nonblock(WOLFSSL* ssl);
int  wolfSSL_dtls_get_current_timeout(WOLFSSL* ssl);
int  wolfSSL_dtls13_use_quick_timeout(WOLFSSL *ssl);
void  wolfSSL_dtls13_set_send_more_acks(WOLFSSL *ssl, int value);

int  wolfSSL_dtls_set_timeout_init(WOLFSSL* ssl, int timeout);

int  wolfSSL_dtls_set_timeout_max(WOLFSSL* ssl, int timeout);

int  wolfSSL_dtls_got_timeout(WOLFSSL* ssl);

int wolfSSL_dtls_retransmit(WOLFSSL* ssl);

int wolfSSL_dtls13_do_scheduled_work(WOLFSSL* ssl);

int wolfSSL_dtls13_pending_work(WOLFSSL* ssl);

int  wolfSSL_dtls(WOLFSSL* ssl);

int  wolfSSL_dtls_set_peer(WOLFSSL* ssl, void* peer, unsigned int peerSz);

int  wolfSSL_dtls_set_pending_peer(WOLFSSL* ssl, void* peer,
                                   unsigned int peerSz);

int  wolfSSL_dtls_get_peer(WOLFSSL* ssl, void* peer, unsigned int* peerSz);

int  wolfSSL_dtls_get0_peer(WOLFSSL* ssl, const void** peer,
```

```
                                unsigned int* peerSz);

char* wolfSSL_ERR_error_string(unsigned long errNumber, char* data);

void  wolfSSL_ERR_error_string_n(unsigned long e, char* buf,
                                                 unsigned long len);

int  wolfSSL_get_shutdown(const WOLFSSL* ssl);

int  wolfSSL_session_reused(WOLFSSL* ssl);

int  wolfSSL_is_init_finished(const WOLFSSL* ssl);

const char*  wolfSSL_get_version(WOLFSSL* ssl);

int  wolfSSL_get_current_cipher_suite(WOLFSSL* ssl);

WOLFSSL_CIPHER*  wolfSSL_get_current_cipher(WOLFSSL* ssl);

const char*  wolfSSL_CIPHER_get_name(const WOLFSSL_CIPHER* cipher);

const char*  wolfSSL_get_cipher(WOLFSSL*);

WOLFSSL_SESSION* wolfSSL_get1_session(WOLFSSL* ssl);

WOLFSSL_METHOD* wolfSSLv23_client_method(void);

int wolfSSL_BIO_get_mem_data(WOLFSSL_BIO* bio,void* p);

void wolfSSL_BIO_set_init(WOLFSSL_BIO* bio, int init);

int wolfSSL_BIO_get_init(WOLFSSL_BIO* bio);

long wolfSSL_BIO_set_fd(WOLFSSL_BIO* b, int fd, int flag);

int wolfSSL_BIO_set_close(WOLFSSL_BIO *b, long flag);

WOLFSSL_BIO_METHOD *wolfSSL_BIO_s_socket(void);

int  wolfSSL_BIO_set_write_buf_size(WOLFSSL_BIO *b, long size);

int  wolfSSL_BIO_make_bio_pair(WOLFSSL_BIO *b1, WOLFSSL_BIO *b2);

int  wolfSSL_BIO_ctrl_reset_read_request(WOLFSSL_BIO *b);

int  wolfSSL_BIO_nread0(WOLFSSL_BIO *bio, char **buf);

int  wolfSSL_BIO_nread(WOLFSSL_BIO *bio, char **buf, int num);

int  wolfSSL_BIO_nwrite(WOLFSSL_BIO *bio, char **buf, int num);

int  wolfSSL_BIO_reset(WOLFSSL_BIO *bio);

int  wolfSSL_BIO_seek(WOLFSSL_BIO *bio, int ofs);
```

```
int  wolfSSL_BIO_write_filename(WOLFSSL_BIO *bio, char *name);

long wolfSSL_BIO_set_mem_eof_return(WOLFSSL_BIO *bio, int v);

long wolfSSL_BIO_get_mem_ptr(WOLFSSL_BIO *bio, WOLFSSL_BUF_MEM **m);

char*       wolfSSL_X509_NAME_oneline(WOLFSSL_X509_NAME* name, char* in, int
 ↪  sz);

WOLFSSL_X509_NAME*  wolfSSL_X509_get_issuer_name(WOLFSSL_X509* cert);

WOLFSSL_X509_NAME*  wolfSSL_X509_get_subject_name(WOLFSSL_X509* cert);

int  wolfSSL_X509_get_isCA(WOLFSSL_X509* x509);

int wolfSSL_X509_NAME_get_text_by_NID(const WOLFSSL_X509_NAME* name, int nid,
                                      char* buf, int len);

int wolfSSL_X509_get_signature_type(WOLFSSL_X509* x509);

void wolfSSL_X509_free(WOLFSSL_X509* x509);

int wolfSSL_X509_get_signature(WOLFSSL_X509* x509, unsigned char* buf, int*
 ↪  bufSz);

int wolfSSL_X509_STORE_add_cert(WOLFSSL_X509_STORE* store, WOLFSSL_X509* x509);

WOLFSSL_STACK* wolfSSL_X509_STORE_CTX_get_chain(
                                                  WOLFSSL_X509_STORE_CTX* ctx);

int wolfSSL_X509_STORE_set_flags(WOLFSSL_X509_STORE* store,
                                                        unsigned long flag);

const byte* wolfSSL_X509_notBefore(WOLFSSL_X509* x509);

const byte* wolfSSL_X509_notAfter(WOLFSSL_X509* x509);

WOLFSSL_BIGNUM *wolfSSL_ASN1_INTEGER_to_BN(const WOLFSSL_ASN1_INTEGER *ai,
                                       WOLFSSL_BIGNUM *bn);

long wolfSSL_CTX_add_extra_chain_cert(WOLFSSL_CTX* ctx, WOLFSSL_X509* x509);

int  wolfSSL_CTX_get_read_ahead(WOLFSSL_CTX* ctx);

int  wolfSSL_CTX_set_read_ahead(WOLFSSL_CTX* ctx, int v);

int  wolfSSL_CTX_set_default_read_buffer_len(WOLFSSL_CTX* ctx, size_t len);

int  wolfSSL_set_default_read_buffer_len(WOLFSSL* ssl, size_t len);

long wolfSSL_CTX_get_default_read_buffer_len(WOLFSSL_CTX* ctx);

long wolfSSL_get_default_read_buffer_len(const WOLFSSL* ssl);
```

```
int wolfSSL_CTX_set_max_send_fragment(WOLFSSL_CTX* ctx, long max_fragment);

int wolfSSL_set_max_send_fragment(WOLFSSL* ssl, long max_fragment);

long wolfSSL_CTX_set_tlsext_status_arg(WOLFSSL_CTX* ctx, void* arg);

void wolfSSL_CTX_set_client_cert_cb(WOLFSSL_CTX *ctx, client_cert_cb cb);

void wolfSSL_CTX_set_cert_cb(WOLFSSL_CTX* ctx, CertSetupCallback cb, void
 ↪  *arg);

int wolfSSL_CTX_set_tlsext_status_cb(WOLFSSL_CTX* ctx, tlsextStatusCb cb);

int wolfSSL_CTX_get_tlsext_status_cb(WOLFSSL_CTX* ctx, tlsextStatusCb* cb);

long wolfSSL_get_tlsext_status_ocsp_resp(WOLFSSL *ssl, unsigned char **resp);

long wolfSSL_set_tlsext_status_ocsp_resp(WOLFSSL *ssl, unsigned char *resp, int
 ↪  len);

int wolfSSL_set_tlsext_status_ocsp_resp_multi(WOLFSSL* ssl, unsigned char
 ↪  *resp, int len, word32 idx);

void wolfSSL_CTX_set_ocsp_status_verify_cb(WOLFSSL_CTX* ctx,
 ↪  ocspVerifyStatusCb cb, void* cbArg);

long wolfSSL_CTX_set_tlsext_opaque_prf_input_callback_arg(
        WOLFSSL_CTX* ctx, void* arg);

long wolfSSL_CTX_set_options(WOLFSSL_CTX* ctx, long opt);

long wolfSSL_set_options(WOLFSSL *s, long op);

long wolfSSL_get_options(const WOLFSSL *s);

long wolfSSL_set_tlsext_debug_arg(WOLFSSL *s, void *arg);

typedef void (*WOLFSSL_TLSEXT_DEBUG_CB)(WOLFSSL* ssl, int client_server,
        int type, const byte* data, int len, void* arg);

long wolfSSL_set_tlsext_debug_callback(WOLFSSL *s, WOLFSSL_TLSEXT_DEBUG_CB cb);

long wolfSSL_set_tlsext_status_type(WOLFSSL *s, int type);

long wolfSSL_get_verify_result(const WOLFSSL *ssl);

void  wolfSSL_ERR_print_errors_fp(XFILE fp, int err);

void  wolfSSL_ERR_print_errors_cb (
        int (*cb)(const char *str, size_t len, void *u), void *u);

void wolfSSL_CTX_set_psk_client_callback(WOLFSSL_CTX* ctx,
                                         wc_psk_client_callback cb);
```

```
void wolfSSL_set_psk_client_callback(WOLFSSL* ssl,
                                                    wc_psk_client_callback cb);

const char* wolfSSL_get_psk_identity_hint(const WOLFSSL*);

const char* wolfSSL_get_psk_identity(const WOLFSSL*);

int wolfSSL_CTX_use_psk_identity_hint(WOLFSSL_CTX* ctx, const char* hint);

int wolfSSL_use_psk_identity_hint(WOLFSSL* ssl, const char* hint);

void wolfSSL_CTX_set_psk_server_callback(WOLFSSL_CTX* ctx,
                                                    wc_psk_server_callback cb);

void wolfSSL_set_psk_server_callback(WOLFSSL* ssl,
                                                    wc_psk_server_callback cb);


int wolfSSL_set_psk_callback_ctx(WOLFSSL* ssl, void* psk_ctx);

int wolfSSL_CTX_set_psk_callback_ctx(WOLFSSL_CTX* ctx, void* psk_ctx);

void* wolfSSL_get_psk_callback_ctx(WOLFSSL* ssl);

void* wolfSSL_CTX_get_psk_callback_ctx(WOLFSSL_CTX* ctx);

int wolfSSL_CTX_allow_anon_cipher(WOLFSSL_CTX* ctx);

WOLFSSL_METHOD *wolfSSLv23_server_method(void);

int  wolfSSL_state(WOLFSSL* ssl);

WOLFSSL_X509* wolfSSL_get_peer_certificate(WOLFSSL* ssl);

int wolfSSL_want_read(WOLFSSL* ssl);

int wolfSSL_want_write(WOLFSSL* ssl);

int wolfSSL_want(WOLFSSL* ssl);

int wolfSSL_check_domain_name(WOLFSSL* ssl, const char* dn);

int wolfSSL_check_ip_address(WOLFSSL* ssl, const char* ipaddr);

int wolfSSL_Init(void);

int wolfSSL_Cleanup(void);

const char* wolfSSL_lib_version(void);

word32 wolfSSL_lib_version_hex(void);

int wolfSSL_negotiate(WOLFSSL* ssl);
```

```
int wolfSSL_set_compression(WOLFSSL* ssl);

int wolfSSL_set_timeout(WOLFSSL* ssl, unsigned int to);

int wolfSSL_CTX_set_timeout(WOLFSSL_CTX* ctx, unsigned int to);

WOLFSSL_X509_CHAIN* wolfSSL_get_peer_chain(WOLFSSL* ssl);

int  wolfSSL_get_chain_count(WOLFSSL_X509_CHAIN* chain);

int  wolfSSL_get_chain_length(WOLFSSL_X509_CHAIN* chain, int idx);

unsigned char* wolfSSL_get_chain_cert(WOLFSSL_X509_CHAIN* chain, int idx);

WOLFSSL_X509* wolfSSL_get_chain_X509(WOLFSSL_X509_CHAIN* chain, int idx);

int  wolfSSL_get_chain_cert_pem(WOLFSSL_X509_CHAIN* chain, int idx,
                                unsigned char* buf, int inLen, int* outLen);

const unsigned char* wolfSSL_get_sessionID(const WOLFSSL_SESSION* s);

int  wolfSSL_X509_get_serial_number(WOLFSSL_X509* x509, unsigned char* in,
                                    int* inOutSz);

char*  wolfSSL_X509_get_subjectCN(WOLFSSL_X509*);

const unsigned char* wolfSSL_X509_get_der(WOLFSSL_X509* x509, int* outSz);

WOLFSSL_ASN1_TIME* wolfSSL_X509_get_notAfter(WOLFSSL_X509*);

int wolfSSL_X509_version(WOLFSSL_X509* x509);

WOLFSSL_X509*
       wolfSSL_X509_d2i_fp(WOLFSSL_X509** x509, FILE* file);

WOLFSSL_X509*
    wolfSSL_X509_load_certificate_file(const char* fname, int format);

unsigned char*
          wolfSSL_X509_get_device_type(WOLFSSL_X509* x509, unsigned char* in,
                                       int* inOutSz);

unsigned char*
          wolfSSL_X509_get_hw_type(WOLFSSL_X509* x509, unsigned char* in,
                                   int* inOutSz);

unsigned char*
          wolfSSL_X509_get_hw_serial_number(WOLFSSL_X509* x509,
                                            unsigned char* in, int* inOutSz);

int  wolfSSL_connect_cert(WOLFSSL* ssl);

WC_PKCS12* wolfSSL_d2i_PKCS12_bio(WOLFSSL_BIO* bio,
```

```
                                        WC_PKCS12** pkcs12);

WC_PKCS12* wolfSSL_i2d_PKCS12_bio(WOLFSSL_BIO* bio,
                                        WC_PKCS12* pkcs12);

int wolfSSL_PKCS12_parse(WC_PKCS12* pkcs12, const char* psw,
     WOLFSSL_EVP_PKEY** pkey, WOLFSSL_X509** cert,
 ↪  WOLF_STACK_OF(WOLFSSL_X509)** ca);

int  wolfSSL_SetTmpDH(WOLFSSL* ssl, const unsigned char* p, int pSz,
                                  const unsigned char* g, int gSz);

int  wolfSSL_SetTmpDH_buffer(WOLFSSL* ssl, const unsigned char* b, long sz,
                                        int format);

int  wolfSSL_SetTmpDH_file(WOLFSSL* ssl, const char* f, int format);

int  wolfSSL_CTX_SetTmpDH(WOLFSSL_CTX* ctx, const unsigned char* p,
                                     int pSz, const unsigned char* g, int gSz);

int  wolfSSL_CTX_SetTmpDH_buffer(WOLFSSL_CTX* ctx, const unsigned char* b,
                                           long sz, int format);

int  wolfSSL_CTX_SetTmpDH_file(WOLFSSL_CTX* ctx, const char* f,
                                            int format);

int wolfSSL_CTX_SetMinDhKey_Sz(WOLFSSL_CTX* ctx, word16 keySz_bits);

int wolfSSL_SetMinDhKey_Sz(WOLFSSL* ssl, word16 keySz_bits);

int wolfSSL_CTX_SetMaxDhKey_Sz(WOLFSSL_CTX* ctx, word16 keySz_bits);

int wolfSSL_SetMaxDhKey_Sz(WOLFSSL* ssl, word16 keySz_bits);

int wolfSSL_GetDhKey_Sz(WOLFSSL* ssl);

int wolfSSL_CTX_SetMinRsaKey_Sz(WOLFSSL_CTX* ctx, short keySz);

int wolfSSL_SetMinRsaKey_Sz(WOLFSSL* ssl, short keySz);

int wolfSSL_CTX_SetMinEccKey_Sz(WOLFSSL_CTX* ctx, short keySz);

int wolfSSL_SetMinEccKey_Sz(WOLFSSL* ssl, short keySz);

int wolfSSL_make_eap_keys(WOLFSSL* ssl, void* key, unsigned int len,
                                                        const char* label);

int wolfSSL_writev(WOLFSSL* ssl, const struct iovec* iov,
                                     int iovcnt);

int wolfSSL_CTX_UnloadCAs(WOLFSSL_CTX* ctx);


int wolfSSL_CTX_UnloadIntermediateCerts(WOLFSSL_CTX* ctx);
```

```
int wolfSSL_CTX_Unload_trust_peers(WOLFSSL_CTX* ctx);

int wolfSSL_CTX_trust_peer_buffer(WOLFSSL_CTX* ctx, const unsigned char* in,
                                  long sz, int format);

int wolfSSL_CTX_load_verify_buffer(WOLFSSL_CTX* ctx, const unsigned char* in,
                                   long sz, int format);


int wolfSSL_CTX_load_verify_buffer_ex(WOLFSSL_CTX* ctx,
                                      const unsigned char* in, long sz,
                                     int format, int userChain, word32 flags);

int wolfSSL_CTX_load_verify_chain_buffer_format(WOLFSSL_CTX* ctx,
                                                const unsigned char* in,
                                                long sz, int format);

int wolfSSL_CTX_use_certificate_buffer(WOLFSSL_CTX* ctx,
                                       const unsigned char* in, long sz,
                                       int format);

int wolfSSL_CTX_use_PrivateKey_buffer(WOLFSSL_CTX* ctx,
                                      const unsigned char* in, long sz,
                                      int format);

int wolfSSL_CTX_use_certificate_chain_buffer(WOLFSSL_CTX* ctx,
                                             const unsigned char* in, long sz);

int wolfSSL_use_certificate_buffer(WOLFSSL* ssl, const unsigned char* in,
                                                long sz, int format);

int wolfSSL_use_PrivateKey_buffer(WOLFSSL* ssl, const unsigned char* in,
                                                long sz, int format);

int wolfSSL_use_certificate_chain_buffer(WOLFSSL* ssl,
                                         const unsigned char* in, long sz);

int wolfSSL_UnloadCertsKeys(WOLFSSL* ssl);

int wolfSSL_CTX_set_group_messages(WOLFSSL_CTX* ctx);

int wolfSSL_set_group_messages(WOLFSSL* ssl);

void wolfSSL_SetFuzzerCb(WOLFSSL* ssl, CallbackFuzzer cbf, void* fCtx);

int   wolfSSL_DTLS_SetCookieSecret(WOLFSSL* ssl,
                                                const byte* secret,
                                                word32 secretSz);

int   wolfSSL_DTLS_SetCookieSecretSecondary(WOLFSSL* ssl,
                                                const byte* secret,
                                                word32 secretSz);
```

```
WC_RNG* wolfSSL_GetRNG(WOLFSSL* ssl);

int wolfSSL_CTX_SetMinVersion(WOLFSSL_CTX* ctx, int version);

int wolfSSL_SetMinVersion(WOLFSSL* ssl, int version);

int wolfSSL_GetObjectSize(void);  /* object size based on build */
int wolfSSL_GetOutputSize(WOLFSSL* ssl, int inSz);

int wolfSSL_GetMaxOutputSize(WOLFSSL* ssl);

int wolfSSL_SetVersion(WOLFSSL* ssl, int version);

void  wolfSSL_CTX_SetMacEncryptCb(WOLFSSL_CTX* ctx, CallbackMacEncrypt cb);

void  wolfSSL_SetMacEncryptCtx(WOLFSSL* ssl, void *ctx);

void* wolfSSL_GetMacEncryptCtx(WOLFSSL* ssl);

void  wolfSSL_CTX_SetDecryptVerifyCb(WOLFSSL_CTX* ctx,
                                     CallbackDecryptVerify cb);

void  wolfSSL_SetDecryptVerifyCtx(WOLFSSL* ssl, void *ctx);

void* wolfSSL_GetDecryptVerifyCtx(WOLFSSL* ssl);

const unsigned char* wolfSSL_GetMacSecret(WOLFSSL* ssl, int verify);

const unsigned char* wolfSSL_GetClientWriteKey(WOLFSSL*);

const unsigned char* wolfSSL_GetClientWriteIV(WOLFSSL*);

const unsigned char* wolfSSL_GetServerWriteKey(WOLFSSL*);

const unsigned char* wolfSSL_GetServerWriteIV(WOLFSSL*);

int                  wolfSSL_GetKeySize(WOLFSSL* ssl);

int                  wolfSSL_GetIVSize(WOLFSSL* ssl);

int                  wolfSSL_GetSide(WOLFSSL* ssl);

int                  wolfSSL_IsTLSv1_1(WOLFSSL* ssl);

int                  wolfSSL_GetBulkCipher(WOLFSSL* ssl);

int                  wolfSSL_GetCipherBlockSize(WOLFSSL* ssl);

int                  wolfSSL_GetAeadMacSize(WOLFSSL* ssl);

int                  wolfSSL_GetHmacSize(WOLFSSL* ssl);

int                  wolfSSL_GetHmacType(WOLFSSL* ssl);
```

```
int                  wolfSSL_GetCipherType(WOLFSSL* ssl);

int wolfSSL_SetTlsHmacInner(WOLFSSL* ssl, byte* inner,
                            word32 sz, int content, int verify);

void  wolfSSL_CTX_SetEccSignCb(WOLFSSL_CTX* ctx, CallbackEccSign cb);

void  wolfSSL_SetEccSignCtx(WOLFSSL* ssl, void *ctx);

void* wolfSSL_GetEccSignCtx(WOLFSSL* ssl);

void  wolfSSL_CTX_SetEccSignCtx(WOLFSSL_CTX* ctx, void *userCtx);

void* wolfSSL_CTX_GetEccSignCtx(WOLFSSL_CTX* ctx);

void  wolfSSL_CTX_SetEccVerifyCb(WOLFSSL_CTX* ctx, CallbackEccVerify cb);

void  wolfSSL_SetEccVerifyCtx(WOLFSSL* ssl, void *ctx);

void* wolfSSL_GetEccVerifyCtx(WOLFSSL* ssl);

void  wolfSSL_CTX_SetRsaSignCb(WOLFSSL_CTX* ctx, CallbackRsaSign cb);

void  wolfSSL_SetRsaSignCtx(WOLFSSL* ssl, void *ctx);

void* wolfSSL_GetRsaSignCtx(WOLFSSL* ssl);

void  wolfSSL_CTX_SetRsaVerifyCb(WOLFSSL_CTX* ctx, CallbackRsaVerify cb);

void  wolfSSL_SetRsaVerifyCtx(WOLFSSL* ssl, void *ctx);

void* wolfSSL_GetRsaVerifyCtx(WOLFSSL* ssl);

void  wolfSSL_CTX_SetRsaEncCb(WOLFSSL_CTX* ctx, CallbackRsaEnc cb);

void  wolfSSL_SetRsaEncCtx(WOLFSSL* ssl, void *ctx);

void* wolfSSL_GetRsaEncCtx(WOLFSSL* ssl);

void  wolfSSL_CTX_SetRsaDecCb(WOLFSSL_CTX* ctx, CallbackRsaDec cb);

void  wolfSSL_SetRsaDecCtx(WOLFSSL* ssl, void *ctx);

void* wolfSSL_GetRsaDecCtx(WOLFSSL* ssl);

void wolfSSL_CTX_SetCACb(WOLFSSL_CTX* ctx, CallbackCACache cb);

WOLFSSL_CERT_MANAGER* wolfSSL_CertManagerNew_ex(void* heap);

WOLFSSL_CERT_MANAGER* wolfSSL_CertManagerNew(void);

void wolfSSL_CertManagerFree(WOLFSSL_CERT_MANAGER* cm);

int wolfSSL_CertManagerLoadCA(WOLFSSL_CERT_MANAGER* cm, const char* f,
```

```
                                                          const char* d);

int wolfSSL_CertManagerLoadCABuffer(WOLFSSL_CERT_MANAGER* cm,
                                    const unsigned char* buff, long sz,
                                    int format);

int wolfSSL_CertManagerUnloadCAs(WOLFSSL_CERT_MANAGER* cm);

int wolfSSL_CertManagerUnloadIntermediateCerts(WOLFSSL_CERT_MANAGER* cm);

int wolfSSL_CertManagerUnload_trust_peers(WOLFSSL_CERT_MANAGER* cm);

int wolfSSL_CertManagerVerify(WOLFSSL_CERT_MANAGER* cm, const char* f,
                                                             int format);

int wolfSSL_CertManagerVerifyBuffer(WOLFSSL_CERT_MANAGER* cm,
                                const unsigned char* buff, long sz, int format);

void wolfSSL_CertManagerSetVerify(WOLFSSL_CERT_MANAGER* cm,
        VerifyCallback verify_callback);

int wolfSSL_CertManagerCheckCRL(WOLFSSL_CERT_MANAGER* cm,
                                const unsigned char* der, int sz);

int wolfSSL_CertManagerEnableCRL(WOLFSSL_CERT_MANAGER* cm,
                                                           int options);

int wolfSSL_CertManagerDisableCRL(WOLFSSL_CERT_MANAGER* cm);

int wolfSSL_CertManagerLoadCRL(WOLFSSL_CERT_MANAGER* cm,
                               const char* path, int type, int monitor);

int wolfSSL_CertManagerLoadCRLBuffer(WOLFSSL_CERT_MANAGER* cm,
                                     const unsigned char* buff, long sz,
                                     int type);

int wolfSSL_CertManagerSetCRL_Cb(WOLFSSL_CERT_MANAGER* cm,
                                 CbMissingCRL cb);

int wolfSSL_CertManagerSetCRLUpdate_Cb(WOLFSSL_CERT_MANAGER* cm,
                                       CbUpdateCRL cb);

int wolfSSL_CertManagerGetCRLInfo(WOLFSSL_CERT_MANAGER* cm, CrlInfo* info,
    const byte* buff, long sz, int type)


int wolfSSL_CertManagerFreeCRL(WOLFSSL_CERT_MANAGER* cm);

int wolfSSL_CertManagerCheckOCSP(WOLFSSL_CERT_MANAGER* cm,
                                 const unsigned char* der, int sz);

int wolfSSL_CertManagerEnableOCSP(WOLFSSL_CERT_MANAGER* cm,
                                                           int options);
```

```
int wolfSSL_CertManagerDisableOCSP(WOLFSSL_CERT_MANAGER* cm);

int wolfSSL_CertManagerSetOCSPOverrideURL(WOLFSSL_CERT_MANAGER* cm,
                                          const char* url);

int wolfSSL_CertManagerSetOCSP_Cb(WOLFSSL_CERT_MANAGER* cm,
                                  CbOCSPIO ioCb, CbOCSPRespFree respFreeCb,
                                  void* ioCbCtx);

int wolfSSL_CertManagerEnableOCSPStapling(
                                                    WOLFSSL_CERT_MANAGER* cm);

int wolfSSL_EnableCRL(WOLFSSL* ssl, int options);

int wolfSSL_DisableCRL(WOLFSSL* ssl);

int wolfSSL_LoadCRL(WOLFSSL* ssl, const char* path, int type, int monitor);

int wolfSSL_SetCRL_Cb(WOLFSSL* ssl, CbMissingCRL cb);

int wolfSSL_EnableOCSP(WOLFSSL* ssl, int options);

int wolfSSL_DisableOCSP(WOLFSSL* ssl);

int wolfSSL_SetOCSP_OverrideURL(WOLFSSL* ssl, const char* url);

int wolfSSL_SetOCSP_Cb(WOLFSSL* ssl, CbOCSPIO ioCb, CbOCSPRespFree respFreeCb,
                       void* ioCbCtx);

int wolfSSL_CTX_EnableCRL(WOLFSSL_CTX* ctx, int options);

int wolfSSL_CTX_DisableCRL(WOLFSSL_CTX* ctx);

int wolfSSL_CTX_LoadCRL(WOLFSSL_CTX* ctx, const char* path, int type, int
 ↪  monitor);

int wolfSSL_CTX_SetCRL_Cb(WOLFSSL_CTX* ctx, CbMissingCRL cb);

int wolfSSL_CTX_EnableOCSP(WOLFSSL_CTX* ctx, int options);

int wolfSSL_CTX_DisableOCSP(WOLFSSL_CTX* ctx);

int wolfSSL_CTX_SetOCSP_OverrideURL(WOLFSSL_CTX* ctx, const char* url);

int wolfSSL_CTX_SetOCSP_Cb(WOLFSSL_CTX* ctx,
                           CbOCSPIO ioCb, CbOCSPRespFree respFreeCb,
                           void* ioCbCtx);

int wolfSSL_CTX_EnableOCSPStapling(WOLFSSL_CTX* ctx);

void wolfSSL_KeepArrays(WOLFSSL* ssl);

void wolfSSL_FreeArrays(WOLFSSL* ssl);
```

```
int wolfSSL_UseSNI(WOLFSSL* ssl, unsigned char type,
                                        const void* data, unsigned short size);

int wolfSSL_CTX_UseSNI(WOLFSSL_CTX* ctx, unsigned char type,
                                        const void* data, unsigned short size);

void wolfSSL_SNI_SetOptions(WOLFSSL* ssl, unsigned char type,
                                                        unsigned char options);

void wolfSSL_CTX_SNI_SetOptions(WOLFSSL_CTX* ctx,
                                    unsigned char type, unsigned char options);

int wolfSSL_SNI_GetFromBuffer(
                 const unsigned char* clientHello, unsigned int helloSz,
                unsigned char type, unsigned char* sni, unsigned int* inOutSz);

unsigned char wolfSSL_SNI_Status(WOLFSSL* ssl, unsigned char type);

unsigned short wolfSSL_SNI_GetRequest(WOLFSSL *ssl,
                                         unsigned char type, void** data);

int wolfSSL_UseALPN(WOLFSSL* ssl, char *protocol_name_list,
                                unsigned int protocol_name_listSz,
                                unsigned char options);

int wolfSSL_ALPN_GetProtocol(WOLFSSL* ssl, char **protocol_name,
                                          unsigned short *size);

int wolfSSL_ALPN_GetPeerProtocol(WOLFSSL* ssl, char **list,
                                             unsigned short *listSz);

int wolfSSL_UseMaxFragment(WOLFSSL* ssl, unsigned char mfl);

int wolfSSL_CTX_UseMaxFragment(WOLFSSL_CTX* ctx, unsigned char mfl);

int wolfSSL_UseTruncatedHMAC(WOLFSSL* ssl);

int wolfSSL_CTX_UseTruncatedHMAC(WOLFSSL_CTX* ctx);

int wolfSSL_UseOCSPStapling(WOLFSSL* ssl,
                            unsigned char status_type, unsigned char options);

int wolfSSL_CTX_UseOCSPStapling(WOLFSSL_CTX* ctx,
                            unsigned char status_type, unsigned char options);

int wolfSSL_UseOCSPStaplingV2(WOLFSSL* ssl,
                            unsigned char status_type, unsigned char options);

int wolfSSL_CTX_UseOCSPStaplingV2(WOLFSSL_CTX* ctx,
                            unsigned char status_type, unsigned char options);

int wolfSSL_UseSupportedCurve(WOLFSSL* ssl, word16 name);

int wolfSSL_CTX_UseSupportedCurve(WOLFSSL_CTX* ctx,
```

```
                                                        word16 name);

int wolfSSL_UseSecureRenegotiation(WOLFSSL* ssl);

int wolfSSL_Rehandshake(WOLFSSL* ssl);

int wolfSSL_UseSessionTicket(WOLFSSL* ssl);

int wolfSSL_CTX_UseSessionTicket(WOLFSSL_CTX* ctx);

int wolfSSL_get_SessionTicket(WOLFSSL* ssl, unsigned char* buf, word32* bufSz);

int wolfSSL_set_SessionTicket(WOLFSSL* ssl, const unsigned char* buf,
                              word32 bufSz);

int wolfSSL_set_SessionTicket_cb(WOLFSSL* ssl,
                                 CallbackSessionTicket cb, void* ctx);

int wolfSSL_send_SessionTicket(WOLFSSL* ssl);

int wolfSSL_CTX_set_TicketEncCb(WOLFSSL_CTX* ctx,
                                          SessionTicketEncCb cb);

int wolfSSL_CTX_set_TicketHint(WOLFSSL_CTX* ctx, int hint);

int wolfSSL_CTX_set_TicketEncCtx(WOLFSSL_CTX* ctx, void* userCtx);

void* wolfSSL_CTX_get_TicketEncCtx(WOLFSSL_CTX* ctx);

int wolfSSL_SetHsDoneCb(WOLFSSL* ssl, HandShakeDoneCb cb, void* user_ctx);

int wolfSSL_PrintSessionStats(void);

int wolfSSL_get_session_stats(unsigned int* active,
                                        unsigned int* total,
                                        unsigned int* peak,
                                        unsigned int* maxSessions);

int wolfSSL_MakeTlsMasterSecret(unsigned char* ms, word32 msLen,
                                const unsigned char* pms, word32 pmsLen,
                               const unsigned char* cr, const unsigned char* sr,
                                int tls1_2, int hash_type);

int wolfSSL_DeriveTlsKeys(unsigned char* key_data, word32 keyLen,
                                const unsigned char* ms, word32 msLen,
                               const unsigned char* sr, const unsigned char* cr,
                                int tls1_2, int hash_type);

int wolfSSL_connect_ex(WOLFSSL* ssl, HandShakeCallBack hsCb,
                       TimeoutCallBack toCb, WOLFSSL_TIMEVAL timeout);

int wolfSSL_accept_ex(WOLFSSL* ssl, HandShakeCallBack hsCb,
                      TimeoutCallBack toCb, WOLFSSL_TIMEVAL timeout);
```

```
long wolfSSL_BIO_set_fp(WOLFSSL_BIO *bio, XFILE fp, int c);

long wolfSSL_BIO_get_fp(WOLFSSL_BIO *bio, XFILE* fp);

int wolfSSL_check_private_key(const WOLFSSL* ssl);

int wolfSSL_X509_get_ext_by_NID(const WOLFSSL_X509 *x, int nid, int lastpos);

void* wolfSSL_X509_get_ext_d2i(const WOLFSSL_X509* x509,
                                                    int nid, int* c, int* idx);

int wolfSSL_X509_digest(const WOLFSSL_X509* x509,
        const WOLFSSL_EVP_MD* digest, unsigned char* buf, unsigned int* len);

int wolfSSL_use_certificate(WOLFSSL* ssl, WOLFSSL_X509* x509);

int wolfSSL_use_certificate_ASN1(WOLFSSL* ssl, const unsigned char* der,
                                 int derSz);

int wolfSSL_use_PrivateKey(WOLFSSL* ssl, WOLFSSL_EVP_PKEY* pkey);

int wolfSSL_use_PrivateKey_ASN1(int pri, WOLFSSL* ssl,
                        const unsigned char* der, long derSz);

int wolfSSL_use_RSAPrivateKey_ASN1(WOLFSSL* ssl, unsigned char* der,
                                                              long derSz);

WOLFSSL_DH *wolfSSL_DSA_dup_DH(const WOLFSSL_DSA *r);

int wolfSSL_SESSION_get_master_key(const WOLFSSL_SESSION* ses,
        unsigned char* out, int outSz);

int wolfSSL_SESSION_get_master_key_length(const WOLFSSL_SESSION* ses);

void wolfSSL_CTX_set_cert_store(WOLFSSL_CTX* ctx,
                                                    WOLFSSL_X509_STORE* str);

WOLFSSL_X509* wolfSSL_d2i_X509_bio(WOLFSSL_BIO* bio, WOLFSSL_X509** x509);

WOLFSSL_X509_STORE* wolfSSL_CTX_get_cert_store(WOLFSSL_CTX* ctx);

size_t wolfSSL_BIO_ctrl_pending(WOLFSSL_BIO *b);

size_t wolfSSL_get_server_random(const WOLFSSL *ssl,
                                           unsigned char *out, size_t outlen);

size_t wolfSSL_get_client_random(const WOLFSSL* ssl,
                                            unsigned char* out, size_t outSz);

wc_pem_password_cb* wolfSSL_CTX_get_default_passwd_cb(WOLFSSL_CTX*
                                                                  ctx);

void *wolfSSL_CTX_get_default_passwd_cb_userdata(WOLFSSL_CTX *ctx);
```

```
WOLFSSL_X509 *wolfSSL_PEM_read_bio_X509_AUX
        (WOLFSSL_BIO *bp, WOLFSSL_X509 **x, wc_pem_password_cb *cb, void *u);

long wolfSSL_CTX_set_tmp_dh(WOLFSSL_CTX* ctx, WOLFSSL_DH* dh);

WOLFSSL_DSA *wolfSSL_PEM_read_bio_DSAparams(WOLFSSL_BIO *bp,
    WOLFSSL_DSA **x, wc_pem_password_cb *cb, void *u);

unsigned long wolfSSL_ERR_peek_last_error(void);

WOLF_STACK_OF(WOLFSSL_X509)* wolfSSL_get_peer_cert_chain(const WOLFSSL*);

long wolfSSL_CTX_clear_options(WOLFSSL_CTX* ctx, long opt);

int wolfSSL_set_jobject(WOLFSSL* ssl, void* objPtr);

void* wolfSSL_get_jobject(WOLFSSL* ssl);

int wolfSSL_set_msg_callback(WOLFSSL *ssl, SSL_Msg_Cb cb);

int wolfSSL_set_msg_callback_arg(WOLFSSL *ssl, void* arg);

char* wolfSSL_X509_get_next_altname(WOLFSSL_X509*);

WOLFSSL_ASN1_TIME* wolfSSL_X509_get_notBefore(WOLFSSL_X509*);

int  wolfSSL_connect(WOLFSSL* ssl);

int  wolfSSL_send_hrr_cookie(WOLFSSL* ssl,
    const unsigned char* secret, unsigned int secretSz);

int  wolfSSL_set_hrr_cookie_secret_secondary(WOLFSSL* ssl,
    const unsigned char* secret, unsigned int secretSz);

int wolfSSL_disable_hrr_cookie(WOLFSSL* ssl);

int  wolfSSL_CTX_no_ticket_TLSv13(WOLFSSL_CTX* ctx);

int  wolfSSL_no_ticket_TLSv13(WOLFSSL* ssl);

int  wolfSSL_CTX_no_dhe_psk(WOLFSSL_CTX* ctx);

int  wolfSSL_no_dhe_psk(WOLFSSL* ssl);

int  wolfSSL_CTX_require_psk(WOLFSSL_CTX* ctx);

int  wolfSSL_require_psk(WOLFSSL* ssl);

int  wolfSSL_update_keys(WOLFSSL* ssl);

int  wolfSSL_key_update_response(WOLFSSL* ssl, int* required);

int  wolfSSL_CTX_allow_post_handshake_auth(WOLFSSL_CTX* ctx);
```

```
int  wolfSSL_allow_post_handshake_auth(WOLFSSL* ssl);

int  wolfSSL_request_certificate(WOLFSSL* ssl);

int  wolfSSL_CTX_set1_sigalgs_list(WOLFSSL_CTX* ctx, const char* list);

int  wolfSSL_set1_sigalgs_list(WOLFSSL* ssl, const char* list);

int  wolfSSL_CTX_set1_groups_list(WOLFSSL_CTX *ctx, const char *list);

int  wolfSSL_set1_groups_list(WOLFSSL *ssl, const char *list);

int  wolfSSL_preferred_group(WOLFSSL* ssl);

int  wolfSSL_CTX_set_groups(WOLFSSL_CTX* ctx, int* groups,
    int count);

int  wolfSSL_set_groups(WOLFSSL* ssl, int* groups, int count);

int  wolfSSL_connect_TLSv13(WOLFSSL* ssl);

wolfSSL_accept_TLSv13(WOLFSSL* ssl);

int  wolfSSL_CTX_set_max_early_data(WOLFSSL_CTX* ctx,
    unsigned int sz);

int  wolfSSL_set_max_early_data(WOLFSSL* ssl, unsigned int sz);

int  wolfSSL_write_early_data(WOLFSSL* ssl, const void* data,
    int sz, int* outSz);

int  wolfSSL_read_early_data(WOLFSSL* ssl, void* data, int sz,
    int* outSz);

int  wolfSSL_CTX_no_early_data_fresh_start_check(WOLFSSL_CTX* ctx);

int wolfSSL_inject(WOLFSSL* ssl, const void* data, int sz);

void wolfSSL_CTX_set_psk_client_tls13_callback(WOLFSSL_CTX* ctx,
    wc_psk_client_tls13_callback cb);

void wolfSSL_set_psk_client_tls13_callback(WOLFSSL* ssl,
    wc_psk_client_tls13_callback cb);

void wolfSSL_CTX_set_psk_server_tls13_callback(WOLFSSL_CTX* ctx,
    wc_psk_server_tls13_callback cb);

void wolfSSL_set_psk_server_tls13_callback(WOLFSSL* ssl,
    wc_psk_server_tls13_callback cb);

    }
    \endcode

    \sa wolfSSL_set_cert_with_extern_psk
```

```
    \sa wolfSSL_CTX_set_psk_client_tls13_callback
    \sa wolfSSL_CTX_set_psk_server_tls13_callback
*/
int wolfSSL_CTX_set_cert_with_extern_psk(WOLFSSL_CTX* ctx, int state);

    }
    \endcode

    \sa wolfSSL_CTX_set_cert_with_extern_psk
    \sa wolfSSL_set_psk_client_tls13_callback
    \sa wolfSSL_set_psk_server_tls13_callback
*/
int wolfSSL_set_cert_with_extern_psk(WOLFSSL* ssl, int state);

int wolfSSL_UseKeyShare(WOLFSSL* ssl, word16 group);

int wolfSSL_NoKeyShares(WOLFSSL* ssl);

WOLFSSL_METHOD *wolfTLSv1_3_server_method_ex(void* heap);

WOLFSSL_METHOD *wolfTLSv1_3_client_method_ex(void* heap);

WOLFSSL_METHOD *wolfTLSv1_3_server_method(void);

WOLFSSL_METHOD *wolfTLSv1_3_client_method(void);

WOLFSSL_METHOD *wolfTLSv1_3_method_ex(void* heap);

WOLFSSL_METHOD *wolfTLSv1_3_method(void);

int wolfSSL_CTX_set_ephemeral_key(WOLFSSL_CTX* ctx, int keyAlgo, const char*
 ↪  key, unsigned int keySz, int format);

int wolfSSL_set_ephemeral_key(WOLFSSL* ssl, int keyAlgo, const char* key,
 ↪  unsigned int keySz, int format);

int wolfSSL_CTX_get_ephemeral_key(WOLFSSL_CTX* ctx, int keyAlgo,
    const unsigned char** key, unsigned int* keySz);

int wolfSSL_get_ephemeral_key(WOLFSSL* ssl, int keyAlgo,
    const unsigned char** key, unsigned int* keySz);

int wolfSSL_RSA_sign_generic_padding(int hashAlg, const unsigned char* hash,
                                     unsigned int hLen, unsigned char* sigRet,
                                     unsigned int* sigLen, WOLFSSL_RSA* rsa,
                                     int flag, int padding);
int wolfSSL_dtls13_has_pending_msg(WOLFSSL *ssl);

unsigned int wolfSSL_SESSION_get_max_early_data(const WOLFSSL_SESSION *s);

int wolfSSL_CRYPTO_get_ex_new_index(int class_index, long argl, void *argp,
                                    WOLFSSL_CRYPTO_EX_new* new_func,
                                    WOLFSSL_CRYPTO_EX_dup* dup_func,
                                    WOLFSSL_CRYPTO_EX_free* free_func);
```

```
int wolfSSL_CTX_set_client_cert_type(WOLFSSL_CTX* ctx, const char* buf, int
 ↪ len);

int wolfSSL_CTX_set_server_cert_type(WOLFSSL_CTX* ctx, const char* buf, int
 ↪ len);

int wolfSSL_set_client_cert_type(WOLFSSL* ssl, const char* buf, int len);

int wolfSSL_set_server_cert_type(WOLFSSL* ssl, const char* buf, int len);

int wolfSSL_CTX_clear_group_messages(WOLFSSL_CTX* ctx);

int wolfSSL_clear_group_messages(WOLFSSL* ssl);

int wolfSSL_get_negotiated_client_cert_type(WOLFSSL* ssl, int* tp);

int wolfSSL_get_negotiated_server_cert_type(WOLFSSL* ssl, int* tp);

int wolfSSL_CTX_set_expected_rpk(WOLFSSL_CTX* ctx, const unsigned char* spki,
    unsigned int spkiSz);

int wolfSSL_set_expected_rpk(WOLFSSL* ssl, const unsigned char* spki,
    unsigned int spkiSz);

int wolfSSL_CTX_clear_expected_rpk(WOLFSSL_CTX* ctx);

int wolfSSL_clear_expected_rpk(WOLFSSL* ssl);

int wolfSSL_dtls_cid_use(WOLFSSL* ssl);

int wolfSSL_dtls_cid_is_enabled(WOLFSSL* ssl);

int wolfSSL_dtls_cid_set(WOLFSSL* ssl, unsigned char* cid,
    unsigned int size);

int wolfSSL_dtls_cid_get_rx_size(WOLFSSL* ssl,
    unsigned int* size);

int wolfSSL_dtls_cid_get_rx(WOLFSSL* ssl, unsigned char* buffer,
    unsigned int bufferSz);

int wolfSSL_dtls_cid_get0_rx(WOLFSSL* ssl, unsigned char** cid);

int wolfSSL_dtls_cid_get_tx_size(WOLFSSL* ssl, unsigned int* size);

int wolfSSL_dtls_cid_get_tx(WOLFSSL* ssl, unsigned char* buffer,
    unsigned int bufferSz);

int wolfSSL_dtls_cid_get0_tx(WOLFSSL* ssl, unsigned char** cid);

int wolfSSL_dtls_cid_max_size(void);

const unsigned char* wolfSSL_dtls_cid_parse(const unsigned char* msg,
```

```
        unsigned int msgSz, unsigned int cidSz);

void wolfSSL_CTX_set_client_CA_list(WOLFSSL_CTX* ctx,
                                    WOLF_STACK_OF(WOLFSSL_X509_NAME)* names);

WOLFSSL_STACK *wolfSSL_CTX_get_client_CA_list(
        const WOLFSSL_CTX *ctx);

void wolfSSL_set_client_CA_list(WOLFSSL* ssl,
                                    WOLF_STACK_OF(WOLFSSL_X509_NAME)* names);

WOLFSSL_STACK* wolfSSL_get_client_CA_list(
            const WOLFSSL* ssl);

void wolfSSL_CTX_set0_CA_list(WOLFSSL_CTX *ctx,
        WOLF_STACK_OF(WOLFSSL_X509_NAME)* names);

WOLFSSL_STACK *wolfSSL_CTX_get0_CA_list(
        const WOLFSSL_CTX *ctx);

void wolfSSL_set0_CA_list(WOLFSSL *ssl,
        WOLF_STACK_OF(WOLFSSL_X509_NAME) *names);

WOLFSSL_STACK *wolfSSL_get0_CA_list(
        const WOLFSSL *ssl);

WOLFSSL_STACK *wolfSSL_get0_peer_CA_list(const WOLFSSL *ssl);

int wolfSSL_get_client_suites_sigalgs(const WOLFSSL* ssl,
        const byte** suites, word16* suiteSz,
        const byte** hashSigAlgo, word16* hashSigAlgoSz);

WOLFSSL_CIPHERSUITE_INFO wolfSSL_get_ciphersuite_info(byte first,
        byte second);

int wolfSSL_get_sigalg_info(byte first, byte second,
        int* hashAlgo, int* sigAlgo);

void wolfSSL_CTX_set_default_passwd_cb(WOLFSSL_CTX* ctx,
                                       wc_pem_password_cb* cb);

void wolfSSL_CTX_set_default_passwd_cb_userdata(WOLFSSL_CTX* ctx,
                                                void* userdata);

int wolfSSL_get_scr_check_enabled(const WOLFSSL* ssl);

int wolfSSL_set_scr_check_enabled(WOLFSSL* ssl, byte enabled);

int wolfSSL_CTX_DisableExtendedMasterSecret(WOLFSSL_CTX* ctx);

int wolfSSL_DisableExtendedMasterSecret(WOLFSSL* ssl);

int wolfSSL_CTX_EnableExtendedMasterSecret(WOLFSSL_CTX* ctx);
```

```
int wolfSSL_EnableExtendedMasterSecret(WOLFSSL* ssl);

int wolfSSL_CTX_RequireExtendedMasterSecret(WOLFSSL_CTX* ctx);

int wolfSSL_RequireExtendedMasterSecret(WOLFSSL* ssl);

int wolfSSL_UseCertificateAuthority(WOLFSSL* ssl,
        const unsigned char* dn, unsigned int dnSz);

int wolfSSL_CTX_UseCertificateAuthority(WOLFSSL_CTX* ctx,
        const unsigned char* dn, unsigned int dnSz);

void wolfSSL_ClearCertificateAuthorities(WOLFSSL* ssl);

void wolfSSL_CTX_ClearCertificateAuthorities(WOLFSSL_CTX* ctx);

int wolfSSL_GetPeerCertificateAuthorityCount(const WOLFSSL* ssl);

int wolfSSL_GetPeerCertificateAuthority(const WOLFSSL* ssl, int idx,
        unsigned char* outDn, unsigned int outDnSz);
```

C.59 tfm.h

C.59.1 Functions

	Name
word32	**CheckRunTimeFastMath**(void)This function checks the runtime fastmath settings for the maximum size of an integer. It is important when a user is using a wolfCrypt library independently, as the FP_SIZE must match for each library in order for math to work correctly. This check is defined as CheckFastMathSettings(), which simply compares CheckRunTimeFastMath and FP_SIZE, returning 0 if there is a mismatch, or 1 if they match.

C.59.2 Functions Documentation

C.59.2.1 function CheckRunTimeFastMath

```
word32 CheckRunTimeFastMath(
    void
)
```

This function checks the runtime fastmath settings for the maximum size of an integer. It is important when a user is using a wolfCrypt library independently, as the FP_SIZE must match for each library in order for math to work correctly. This check is defined as CheckFastMathSettings(), which simply compares CheckRunTimeFastMath and FP_SIZE, returning 0 if there is a mismatch, or 1 if they match.

Parameters:

- **none** No parameters.

See: CheckRunTimeSettings

Return: FP_SIZE Returns FP_SIZE, corresponding to the max size available for the math library.

Example

```
if (CheckFastMathSettings() != 1) {
return err_sys("Build vs. runtime fastmath FP_MAX_BITS mismatch\n");
}
// This is converted by the preprocessor to:
// if ( (CheckRunTimeFastMath() == FP_SIZE) != 1) {
// and confirms that the fast math settings match
// the compile time settings
```

C.59.3 Source code

```
word32 CheckRunTimeFastMath(void);
```

C.60 types.h

C.60.1 Functions

	Name
void *	**XMALLOC**(size_t n, void * heap, int type)This is not actually a function, but rather a preprocessor macro, which allows the user to substitute in their own malloc, realloc, and free functions in place of the standard C memory functions. To use external memory functions, define XMALLOC_USER. This will cause the memory functions to be replaced by external functions of the form: extern void *XMALLOC(size_t n, void* heap, int type); extern void _XREALLOC(void *p, size_t n, void_ heap, int type); extern void XFREE(void* p, void *heap, int type); To use the basic C memory functions in place of wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free, define NO_WOLFSSL_MEMORY. This will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) free((xp));} #define XREALLOC(p, n, h, t) realloc((p), (n)) If none of these options are selected, the system will default to use the wolfSSL memory functions. A user can set custom memory functions through callback hooks, (see wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free). This option will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, wolfSSL_Malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) wolfSSL_Free((xp));} #define XREALLOC(p, n, h, t) wolfSSL_Realloc((p), (n))

	Name
void *	**XREALLOC**(void * p, size_t n, void * heap, int type)This is not actually a function, but rather a preprocessor macro, which allows the user to substitute in their own malloc, realloc, and free functions in place of the standard C memory functions. To use external memory functions, define XMALLOC_USER. This will cause the memory functions to be replaced by external functions of the form: extern void *XMALLOC(size_t n, void* heap, int type); extern void _XREALLOC(void *p, size_t n, void_ heap, int type); extern void XFREE(void* p, void *heap, int type); To use the basic C memory functions in place of wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free, define NO_WOLFSSL_MEMORY. This will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) free((xp));} #define XREALLOC(p, n, h, t) realloc((p), (n)) If none of these options are selected, the system will default to use the wolfSSL memory functions. A user can set custom memory functions through callback hooks, (see wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free). This option will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, wolfSSL_Malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) wolfSSL_Free((xp));} #define XREALLOC(p, n, h, t) wolfSSL_Realloc((p), (n))

COPYRIGHT ©2024 wolfSSL Inc.

	Name
void	**XFREE**(void * p, void * heap, int type)This is not actually a function, but rather a preprocessor macro, which allows the user to substitute in their own malloc, realloc, and free functions in place of the standard C memory functions. To use external memory functions, define XMALLOC_USER. This will cause the memory functions to be replaced by external functions of the form: extern void *XMALLOC(size_t n, void* heap, int type); extern void _XREALLOC(void *p, size_t n, void_ heap, int type); extern void XFREE(void* p, void *heap, int type); To use the basic C memory functions in place of wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free, define NO_WOLFSSL_MEMORY. This will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) free((xp));} #define XREALLOC(p, n, h, t) realloc((p), (n)) If none of these options are selected, the system will default to use the wolfSSL memory functions. A user can set custom memory functions through callback hooks, (see wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free). This option will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, wolfSSL_Malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) wolfSSL_Free((xp));} #define XREALLOC(p, n, h, t) wolfSSL_Realloc((p), (n))
word32	**CheckRunTimeSettings**(void)This function checks the compile time class settings. It is important when a user is using a wolfCrypt library independently, as the settings must match between libraries for math to work correctly. This check is defined as CheckCtcSettings(), which simply compares CheckRunTimeSettings and CTC_SETTINGS, returning 0 if there is a mismatch, or 1 if they match.
char *	**wc_strtok**(char * str, const char * delim, char ** nextp)Thread-safe string tokenization.
char *	**wc_strsep**(char ** stringp, const char * delim)Separates string by delimiter.
size_t	**wc_strlcpy**(char * dst, const char * src, size_t dstSize)Safely copies string with size limit.

	Name
int	**wc_UnpackOctets**(byte * out, word32 outSz, const byte * in, word32 inSz, word32 octetSz)Expands a packed octet stream into the one_octet_per_byte_cell form every wolfCrypt byte* API expects. Declared only when WOLFSSL_WIDE_BYTE is set (CHAR_BIT != 8); elsewhere the two layouts coincide and no conversion is needed.
int	**wc_PackOctets**(byte * out, word32 outSz, const byte * in, word32 inSz, word32 octetSz)Packs a one_octet_per_byte_cell buffer into an octet stream, for storing to flash or handing to a byte_oriented peripheral. Inverse of wc_UnpackOctets(); requires WOLFSSL_WIDE_BYTE.
size_t	**wc_strlcat**(char * dst, const char * src, size_t dstSize)Safely concatenates strings with size limit.
int	**wc_strcasecmp**(const char * s1, const char * s2)Case-insensitive string comparison.
int	**wc_strncasecmp**(const char * s1, const char * s2, size_t n)Case-insensitive string comparison with length limit.
int	**wolfSSL_NewThread**(THREAD_TYPE * thread, THREAD_CB cb, void * arg)Creates a new thread.
int	**wolfSSL_NewThreadNoJoin**(THREAD_CB_NOJOIN cb, void * arg)Creates a detached thread.
int	**wolfSSL_JoinThread**(THREAD_TYPE thread)Waits for thread to complete.
int	**wolfSSL_CondInit**(COND_TYPE * cond)Initializes condition variable.
int	**wolfSSL_CondFree**(COND_TYPE * cond)Frees condition variable.
int	**wolfSSL_CondSignal**(COND_TYPE * cond)Signals condition variable.
int	**wolfSSL_CondWait**(COND_TYPE * cond)Waits on condition variable.
int	**wolfSSL_CondStart**(COND_TYPE * cond)Starts condition variable.
int	**wolfSSL_CondEnd**(COND_TYPE * cond)Ends condition variable.

C.60.2 Functions Documentation

C.60.2.1 function XMALLOC

```
void * XMALLOC(
    size_t n,
    void * heap,
    int type
)
```

This is not actually a function, but rather a preprocessor macro, which allows the user to substitute in their own malloc, realloc, and free functions in place of the standard C memory functions. To use external memory functions, define XMALLOC_USER. This will cause the memory functions to be replaced by external functions of the form: extern void *XMALLOC(size_t n, void* heap, int type); extern void _XREALLOC(void *p, size_t n, void_ heap, int type); extern void XFREE(void* p, void *heap, int type); To use the basic C memory functions in place of wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free, define NO_WOLFSSL_MEMORY. This will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) free((xp));} #define XREALLOC(p, n, h, t) realloc((p), (n)) If none of these options are selected, the system will default to use the wolfSSL memory functions. A user can set custom memory functions through callback hooks, (see wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free). This option will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, wolfSSL_Malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) wolfSSL_Free((xp));} #define XREALLOC(p, n, h, t) wolfSSL_Realloc((p), (n))

Parameters:

- **s** size of memory to allocate
- **h** (used by custom XMALLOC function) pointer to the heap to use
- **t** memory allocation types for user hints. See enum in types.h

See:

- wolfSSL_Malloc
- wolfSSL_Realloc
- wolfSSL_Free
- wolfSSL_SetAllocators

Return:

- pointer Return a pointer to allocated memory on success
- NULL on failure

Example

```
int* tenInts = XMALLOC(sizeof(int)*10, NULL, DYNAMIC_TYPE_TMP_BUFFER);
if (tenInts == NULL) {
    // error allocating space
    return MEMORY_E;
}
```

C.60.2.2 function XREALLOC

```
void * XREALLOC(
    void * p,
    size_t n,
    void * heap,
    int type
)
```

This is not actually a function, but rather a preprocessor macro, which allows the user to substitute in their own malloc, realloc, and free functions in place of the standard C memory functions. To use external memory functions, define XMALLOC_USER. This will cause the memory functions to be replaced by external functions of the form: extern void *XMALLOC(size_t n, void* heap, int type); extern void _XREALLOC(void *p, size_t n, void_ heap, int type); extern void XFREE(void* p, void *heap, int type); To use the basic C memory functions in place of wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free, define NO_WOLFSSL_MEMORY. This will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) free((xp));} #define XREALLOC(p, n, h, t) realloc((p), (n)) If none of these options are selected, the system will default to use the wolfSSL memory functions. A user can set custom memory functions through callback hooks, (see wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free).

This option will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, wolfSSL_Malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) wolfSSL_Free((xp));} #define XREALLOC(p, n, h, t) wolfSSL_Realloc((p), (n))

Parameters:

- **p** pointer to the address to reallocate
- **n** size of memory to allocate
- **h** (used by custom XREALLOC function) pointer to the heap to use
- **t** memory allocation types for user hints. See enum in types.h

See:

- wolfSSL_Malloc
- wolfSSL_Realloc
- wolfSSL_Free
- wolfSSL_SetAllocators

Return:

- Return a pointer to allocated memory on success
- NULL on failure

Example

```
int* tenInts = (int*)XMALLOC(sizeof(int)*10, NULL, DYNAMIC_TYPE_TMP_BUFFER);
int* twentyInts = (int*)XREALLOC(tenInts, sizeof(int)*20, NULL,
    DYNAMIC_TYPE_TMP_BUFFER);
```

C.60.2.3 function XFREE

```
void XFREE(
    void * p,
    void * heap,
    int type
)
```

This is not actually a function, but rather a preprocessor macro, which allows the user to substitute in their own malloc, realloc, and free functions in place of the standard C memory functions. To use external memory functions, define XMALLOC_USER. This will cause the memory functions to be replaced by external functions of the form: extern void *XMALLOC(size_t n, void* heap, int type); extern void _XREALLOC(void *p, size_t n, void_ heap, int type); extern void XFREE(void* p, void *heap, int type); To use the basic C memory functions in place of wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free, define NO_WOLFSSL_MEMORY. This will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) free((xp));} #define XREALLOC(p, n, h, t) realloc((p), (n)) If none of these options are selected, the system will default to use the wolfSSL memory functions. A user can set custom memory functions through callback hooks, (see wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free). This option will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, wolfSSL_Malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) wolfSSL_Free((xp));} #define XREALLOC(p, n, h, t) wolfSSL_Realloc((p), (n))

Parameters:

- **p** pointer to the address to free
- **h** (used by custom XFREE function) pointer to the heap to use
- **t** memory allocation types for user hints. See enum in types.h

See:

- wolfSSL_Malloc
- wolfSSL_Realloc

- wolfSSL_Free
- wolfSSL_SetAllocators

Return: none No returns.

Example

```
int* tenInts = XMALLOC(sizeof(int) * 10, NULL, DYNAMIC_TYPE_TMP_BUFFER);
if (tenInts == NULL) {
    // error allocating space
    return MEMORY_E;
}
```

C.60.2.4 function CheckRunTimeSettings

```
word32 CheckRunTimeSettings(
    void
)
```

This function checks the compile time class settings. It is important when a user is using a wolfCrypt library independently, as the settings must match between libraries for math to work correctly. This check is defined as CheckCtcSettings(), which simply compares CheckRunTimeSettings and CTC_SETTINGS, returning 0 if there is a mismatch, or 1 if they match.

Parameters:

- **none** No Parameters.

See: CheckRunTimeFastMath

Return: settings Returns the runtime CTC_SETTINGS (Compile Time Settings)

Example

```
if (CheckCtcSettings() != 1) {
    return err_sys("Build vs. runtime math mismatch\n");
}
// This is converted by the preprocessor to:
// if ( (CheckCtcSettings() == CTC_SETTINGS) != 1) {
// and will compare whether the compile time class settings
// match the current settings
```

C.60.2.5 function wc_strtok

```
char * wc_strtok(
    char * str,
    const char * delim,
    char ** nextp
)
```

Thread-safe string tokenization.

Parameters:

- **str** String to tokenize (NULL for continuation)
- **delim** Delimiter characters
- **nextp** Pointer to save position

See: wc_strsep

Return: Pointer to next token or NULL

Example

```
char str[] = "one,two,three";
char* saveptr;
char* token = wc_strtok(str, ",", &saveptr);
```

C.60.2.6 function wc_strsep

```
char * wc_strsep(
    char ** stringp,
    const char * delim
)
```

Separates string by delimiter.

Parameters:

- **stringp** Pointer to string pointer
- **delim** Delimiter characters

See: wc_strtok

Return: Pointer to token or NULL

Example

```
char str[] = "one,two,three";
char* ptr = str;
char* token = wc_strsep(&ptr, ",");
```

C.60.2.7 function wc_strlcpy

```
size_t wc_strlcpy(
    char * dst,
    const char * src,
    size_t dstSize
)
```

Safely copies string with size limit.

Parameters:

- **dst** Destination buffer
- **src** Source string
- **dstSize** Destination buffer size

See: wc_strlcat

Return: Length of source string

Example

```
char dst[10];
size_t len = wc_strlcpy(dst, "hello", sizeof(dst));
```

C.60.2.8 function wc_UnpackOctets

```
int wc_UnpackOctets(
    byte * out,
    word32 outSz,
    const byte * in,
    word32 inSz,
```

```
    word32 octetSz
)
```

Expands a packed octet stream into the one-octet-per-byte-cell form every wolfCrypt byte* API expects. Declared only when WOLFSSL_WIDE_BYTE is set (CHAR_BIT != 8); elsewhere the two layouts coincide and no conversion is needed.

Parameters:

- **out** Destination, one octet per cell
- **outSz** Size of out in byte cells
- **in** Packed source, WC_OCTETS_PER_BYTE octets per cell, low octet first, and must not overlap out
- **inSz** Size of in in byte cells; must be at least WC_PACKED_CELLS(octetSz)
- **octetSz** Number of octets to expand

See: wc_PackOctets

Return:

- 0 on success
- BAD_FUNC_ARG when out or in is NULL
- BUFFER_E when out is smaller than octetSz, or in smaller than WC_PACKED_CELLS(octetSz)

Example

```
byte sig[WC_MLDSA_65_SIG_SIZE];
int ret = wc_UnpackOctets(sig, (word32)sizeof(sig), sigPacked,
                          (word32)sizeof(sigPacked),
                          WC_MLDSA_65_SIG_SIZE);
```

C.60.2.9 function wc_PackOctets

```
int wc_PackOctets(
    byte * out,
    word32 outSz,
    const byte * in,
    word32 inSz,
    word32 octetSz
)
```

Packs a one-octet-per-byte-cell buffer into an octet stream, for storing to flash or handing to a byte-oriented peripheral. Inverse of wc_UnpackOctets(); requires WOLFSSL_WIDE_BYTE.

Parameters:

- **out** Destination for the packed stream
- **outSz** Size of out in byte cells
- **in** Source, one octet per cell, and must not overlap out
- **inSz** Size of in in byte cells; must be at least octetSz
- **octetSz** Number of octets to pack

See: wc_UnpackOctets

Return:

- 0 on success
- BAD_FUNC_ARG when out or in is NULL
- BUFFER_E when out is smaller than WC_PACKED_CELLS(octetSz), or in smaller than octetSz

Example

```
byte packed[WC_PACKED_CELLS(sizeof(sig))];
int ret = wc_PackOctets(packed, (word32)sizeof(packed), sig,
                        (word32)sizeof(sig), (word32)sizeof(sig));
```

C.60.2.10 function wc_strlcat

```
size_t wc_strlcat(
    char * dst,
    const char * src,
    size_t dstSize
)
```

Safely concatenates strings with size limit.

Parameters:

- **dst** Destination buffer
- **src** Source string
- **dstSize** Destination buffer size

See: wc_strlcpy

Return: Total length attempted

Example

```
char dst[20] = "hello";
size_t len = wc_strlcat(dst, " world", sizeof(dst));
```

C.60.2.11 function wc_strcasecmp

```
int wc_strcasecmp(
    const char * s1,
    const char * s2
)
```

Case-insensitive string comparison.

Parameters:

- **s1** First string
- **s2** Second string

See: wc_strncasecmp

Return: 0 if equal, non-zero otherwise

Example

```
if (wc_strcasecmp("Hello", "hello") == 0) {
    // strings are equal
}
```

C.60.2.12 function wc_strncasecmp

```
int wc_strncasecmp(
    const char * s1,
    const char * s2,
    size_t n
)
```

Case-insensitive string comparison with length limit.

Parameters:

- **s1** First string
- **s2** Second string
- **n** Maximum characters to compare

See: wc_strcasecmp

Return: 0 if equal, non-zero otherwise

Example

```
if (wc_strncasecmp("Hello", "hello", 5) == 0) {
    // strings are equal
}
```

C.60.2.13 function wolfSSL_NewThread

```
int wolfSSL_NewThread(
    THREAD_TYPE * thread,
    THREAD_CB cb,
    void * arg
)
```

Creates a new thread.

Parameters:

- **thread** Thread handle pointer
- **cb** Thread callback function
- **arg** Argument to pass to callback

See: wolfSSL_JoinThread

Return:

- 0 on success
- negative on error

Example

```
THREAD_TYPE thread;
int ret = wolfSSL_NewThread(&thread, myCallback, NULL);
```

C.60.2.14 function wolfSSL_NewThreadNoJoin

```
int wolfSSL_NewThreadNoJoin(
    THREAD_CB_NOJOIN cb,
    void * arg
)
```

Creates a detached thread.

Parameters:

- **cb** Thread callback function
- **arg** Argument to pass to callback

See: wolfSSL_NewThread

Return:

- 0 on success
- negative on error

Example

```
int ret = wolfSSL_NewThreadNoJoin(myCallback, NULL);
```

C.60.2.15 function wolfSSL_JoinThread

```
int wolfSSL_JoinThread(
    THREAD_TYPE thread
)
```

Waits for thread to complete.

Parameters:

- **thread** Thread handle

See: wolfSSL_NewThread

Return:

- 0 on success
- negative on error

Example

```
THREAD_TYPE thread;
wolfSSL_NewThread(&thread, myCallback, NULL);
int ret = wolfSSL_JoinThread(thread);
```

C.60.2.16 function wolfSSL_CondInit

```
int wolfSSL_CondInit(
    COND_TYPE * cond
)
```

Initializes condition variable.

Parameters:

- **cond** Condition variable pointer

See: wolfSSL_CondFree

Return:

- 0 on success
- negative on error

Example

```
COND_TYPE cond;
int ret = wolfSSL_CondInit(&cond);
```

C.60.2.17 function wolfSSL_CondFree

```
int wolfSSL_CondFree(
    COND_TYPE * cond
)
```

Frees condition variable.

Parameters:

- **cond** Condition variable pointer

See: wolfSSL_CondInit

Return:

- 0 on success
- negative on error

Example

```
COND_TYPE cond;
wolfSSL_CondInit(&cond);
int ret = wolfSSL_CondFree(&cond);
```

C.60.2.18 function wolfSSL_CondSignal

```
int wolfSSL_CondSignal(
    COND_TYPE * cond
)
```

Signals condition variable.

Parameters:

- **cond** Condition variable pointer

See: wolfSSL_CondWait

Return:

- 0 on success
- negative on error

Example

```
COND_TYPE cond;
int ret = wolfSSL_CondSignal(&cond);
```

C.60.2.19 function wolfSSL_CondWait

```
int wolfSSL_CondWait(
    COND_TYPE * cond
)
```

Waits on condition variable.

Parameters:

- **cond** Condition variable pointer

See: wolfSSL_CondSignal

Return:

- 0 on success
- negative on error

Example

```
COND_TYPE cond;
int ret = wolfSSL_CondWait(&cond);
```

C.60.2.20 function wolfSSL_CondStart

```
int wolfSSL_CondStart(
    COND_TYPE * cond
)
```

Starts condition variable.

Parameters:

- **cond** Condition variable pointer

See: wolfSSL_CondEnd

Return:

- 0 on success
- negative on error

Example

```
COND_TYPE cond;
int ret = wolfSSL_CondStart(&cond);
```

C.60.2.21 function wolfSSL_CondEnd

```
int wolfSSL_CondEnd(
    COND_TYPE * cond
)
```

Ends condition variable.

Parameters:

- **cond** Condition variable pointer

See: wolfSSL_CondStart

Return:

- 0 on success
- negative on error

Example

```
COND_TYPE cond;
wolfSSL_CondStart(&cond);
int ret = wolfSSL_CondEnd(&cond);
```

C.60.3 Source code

```
void* XMALLOC(size_t n, void* heap, int type);

void* XREALLOC(void *p, size_t n, void* heap, int type);

void XFREE(void *p, void* heap, int type);

word32 CheckRunTimeSettings(void);

char* wc_strtok(char *str, const char *delim, char **nextp);

char* wc_strsep(char **stringp, const char *delim);
```

```
size_t wc_strlcpy(char *dst, const char *src, size_t dstSize);

int wc_UnpackOctets(byte* out, word32 outSz, const byte* in, word32 inSz,
                    word32 octetSz);

int wc_PackOctets(byte* out, word32 outSz, const byte* in, word32 inSz,
                  word32 octetSz);

size_t wc_strlcat(char *dst, const char *src, size_t dstSize);

int wc_strcasecmp(const char *s1, const char *s2);

int wc_strncasecmp(const char *s1, const char *s2, size_t n);

int wolfSSL_NewThread(THREAD_TYPE* thread, THREAD_CB cb, void* arg);

int wolfSSL_NewThreadNoJoin(THREAD_CB_NOJOIN cb, void* arg);

int wolfSSL_JoinThread(THREAD_TYPE thread);

int wolfSSL_CondInit(COND_TYPE* cond);

int wolfSSL_CondFree(COND_TYPE* cond);

int wolfSSL_CondSignal(COND_TYPE* cond);

int wolfSSL_CondWait(COND_TYPE* cond);

int wolfSSL_CondStart(COND_TYPE* cond);

int wolfSSL_CondEnd(COND_TYPE* cond);
```

C.61 wc_encrypt.h

C.61.1 Functions

	Name
int	**wc_AesCbcDecryptWithKey**(byte * out, const byte * in, word32 inSz, const byte * key, word32 keySz, const byte * iv)Decrypts a cipher from the input buffer in, and places the resulting plain text in the output buffer out using cipher block chaining with AES. This function does not require an AES structure to be initialized. Instead, it takes in a key and an iv (initialization vector) and uses these to initialize an AES object and then decrypt the cipher text.

	Name
int	**wc_Des_CbcDecryptWithKey**(byte * out, const byte * in, word32 sz, const byte * key, const byte * iv)This function decrypts the input ciphertext, in, and stores the resulting plaintext in the output buffer, out. It uses DES encryption with cipher block chaining (CBC) mode. This function is a substitute for wc_Des_CbcDecrypt, allowing the user to decrypt a message without directly instantiating a Des structure.
int	**wc_Des_CbcEncryptWithKey**(byte * out, const byte * in, word32 sz, const byte * key, const byte * iv)This function encrypts the input plaintext, in, and stores the resulting ciphertext in the output buffer, out. It uses DES encryption with cipher block chaining (CBC) mode. This function is a substitute for wc_Des_CbcEncrypt, allowing the user to encrypt a message without directly instantiating a Des structure.
int	**wc_Des3_CbcEncryptWithKey**(byte * out, const byte * in, word32 sz, const byte * key, const byte * iv)This function encrypts the input plaintext, in, and stores the resulting ciphertext in the output buffer, out. It uses Triple DES (3DES) encryption with cipher block chaining (CBC) mode. This function is a substitute for wc_Des3_CbcEncrypt, allowing the user to encrypt a message without directly instantiating a Des3 structure.
int	**wc_Des3_CbcDecryptWithKey**(byte * out, const byte * in, word32 sz, const byte * key, const byte * iv)This function decrypts the input ciphertext, in, and stores the resulting plaintext in the output buffer, out. It uses Triple Des (3DES) encryption with cipher block chaining (CBC) mode. This function is a substitute for wc_Des3_CbcDecrypt, allowing the user to decrypt a message without directly instantiating a Des3 structure.
int	**wc_AesCbcEncryptWithKey**(byte * out, const byte * in, word32 inSz, const byte * key, word32 keySz, const byte * iv)This function encrypts a plaintext message and stores the result in the output buffer. It uses AES encryption with cipher block chaining (CBC) mode. This function does not require an AES structure to be initialized. Instead, it takes in a key and an iv and uses these to encrypt the message.

	Name
int	**wc_BufferKeyDecrypt**(struct EncryptedInfo * info, byte * der, word32 derSz, const byte * password, int passwordSz, int hashType)This function decrypts an encrypted key buffer using the provided password. It supports encryption algorithms DES, 3DES, and AES. The encryption information is provided in the EncryptedInfo structure.
int	**wc_BufferKeyEncrypt**(struct EncryptedInfo * info, byte * der, word32 derSz, const byte * password, int passwordSz, int hashType)This function encrypts a key buffer using the provided password. It supports encryption algorithms DES, 3DES, and AES. The encryption information is provided in the EncryptedInfo structure.

C.61.2 Functions Documentation

C.61.2.1 function wc_AesCbcDecryptWithKey

```
int wc_AesCbcDecryptWithKey(
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * key,
    word32 keySz,
    const byte * iv
)
```

Decrypts a cipher from the input buffer in, and places the resulting plain text in the output buffer out using cipher block chaining with AES. This function does not require an AES structure to be initialized. Instead, it takes in a key and an iv (initialization vector) and uses these to initialize an AES object and then decrypt the cipher text.

Parameters:

- **out** pointer to the output buffer in which to store the plain text of the decrypted message
- **in** pointer to the input buffer containing cipher text to be decrypted
- **inSz** size of input message
- **key** 16, 24, or 32 byte secret key for decryption
- **keySz** size of key used for decryption

See:

- wc_AesSetKey
- wc_AesSetIV
- wc_AesCbcEncrypt
- wc_AesCbcDecrypt

Return:

- 0 On successfully decrypting message
- BAD_ALIGN_E Returned on block align error
- BAD_FUNC_ARG Returned if key length is invalid or AES object is null during AesSetIV

- MEMORY_E Returned if WOLFSSL_SMALL_STACK is enabled and XMALLOC fails to instantiate an AES object.

Example

```
int ret = 0;
byte key[] = { some 16, 24, or 32 byte key };
byte iv[]  = { some 16 byte iv };
byte cipher[AES_BLOCK_SIZE * n]; //n being a positive integer making
cipher some multiple of 16 bytes
// fill cipher with cipher text
byte plain [AES_BLOCK_SIZE * n];
if ((ret = wc_AesCbcDecryptWithKey(plain, cipher, AES_BLOCK_SIZE, key,
AES_BLOCK_SIZE, iv)) != 0 ) {
// Decrypt Error
}
```

C.61.2.2 function wc_Des_CbcDecryptWithKey

```
int wc_Des_CbcDecryptWithKey(
    byte * out,
    const byte * in,
    word32 sz,
    const byte * key,
    const byte * iv
)
```

This function decrypts the input ciphertext, in, and stores the resulting plaintext in the output buffer, out. It uses DES encryption with cipher block chaining (CBC) mode. This function is a substitute for wc_Des_CbcDecrypt, allowing the user to decrypt a message without directly instantiating a Des structure.

Parameters:

- **out** pointer to the buffer in which to store the decrypted plaintext
- **in** pointer to the input buffer containing the encrypted ciphertext
- **sz** length of the ciphertext to decrypt
- **key** pointer to the buffer containing the 8 byte key to use for decryption
- **iv** pointer to the buffer containing the 8 byte iv to use for decryption. If no iv is provided, the iv defaults to 0

See: wc_Des_CbcDecrypt

Return:

- 0 Returned upon successfully decrypting the given ciphertext
- MEMORY_E Returned if there is an error allocating space for a Des structure

3

Example

```
int ret;
byte key[] = { // initialize with 8 byte key };
byte iv[]  = { // initialize with 8 byte iv };

byte cipher[]  = { // initialize with ciphertext };
byte decoded[sizeof(cipher)];

if ( wc_Des_CbcDecryptWithKey(decoded, cipher, sizeof(cipher), key,
```

```
iv) != 0) {
    // error decrypting message
}
```

C.61.2.3 function wc_Des_CbcEncryptWithKey

```
int wc_Des_CbcEncryptWithKey(
    byte * out,
    const byte * in,
    word32 sz,
    const byte * key,
    const byte * iv
)
```

This function encrypts the input plaintext, in, and stores the resulting ciphertext in the output buffer, out. It uses DES encryption with cipher block chaining (CBC) mode. This function is a substitute for wc_Des_CbcEncrypt, allowing the user to encrypt a message without directly instantiating a Des structure.

Parameters:

- **out** Final encrypted data
- **in** Data to be encrypted, must be padded to Des block size.
- **sz** Size of input buffer.
- **key** Pointer to the key to use for encryption.
- **iv** Initialization vector

See:

- wc_Des_CbcDecryptWithKey
- wc_Des_CbcEncrypt

Return:

- 0 Returned after successfully encrypting data.
- MEMORY_E Returned if there's an error allocating memory for a Des structure.
- <0 Returned on any error during encryption.

3

Example

```
byte key[] = { // initialize with 8 byte key };
byte iv[]  = { // initialize with 8 byte iv };
byte in[] = { // Initialize with plaintext };
byte out[sizeof(in)];
if ( wc_Des_CbcEncryptWithKey(&out, in, sizeof(in), key, iv) != 0)
{
    // error encrypting message
}
```

C.61.2.4 function wc_Des3_CbcEncryptWithKey

```
int wc_Des3_CbcEncryptWithKey(
    byte * out,
    const byte * in,
    word32 sz,
    const byte * key,
```

```
    const byte * iv
)
```

This function encrypts the input plaintext, in, and stores the resulting ciphertext in the output buffer, out. It uses Triple DES (3DES) encryption with cipher block chaining (CBC) mode. This function is a substitute for wc_Des3_CbcEncrypt, allowing the user to encrypt a message without directly instantiating a Des3 structure.

Parameters:

- **out** Final encrypted data
- **in** Data to be encrypted, must be padded to Des block size.
- **sz** Size of input buffer.
- **key** Pointer to the key to use for encryption.
- **iv** Initialization vector

See:

- wc_Des3_CbcDecryptWithKey
- wc_Des_CbcEncryptWithKey
- wc_Des_CbcDecryptWithKey

Return:

- 0 Returned after successfully encrypting data.
- MEMORY_E Returned if there's an error allocating memory for a Des structure.
- <0 Returned on any error during encryption.

3

Example

```
byte key[] = { // initialize with 8 byte key };
byte iv[]  = { // initialize with 8 byte iv };

byte in[] = { // Initialize with plaintext };
byte out[sizeof(in)];

if ( wc_Des3_CbcEncryptWithKey(&out, in, sizeof(in), key, iv) != 0)
{
    // error encrypting message
}
```

C.61.2.5 function wc_Des3_CbcDecryptWithKey

```
int wc_Des3_CbcDecryptWithKey(
    byte * out,
    const byte * in,
    word32 sz,
    const byte * key,
    const byte * iv
)
```

This function decrypts the input ciphertext, in, and stores the resulting plaintext in the output buffer, out. It uses Triple Des (3DES) encryption with cipher block chaining (CBC) mode. This function is a substitute for wc_Des3_CbcDecrypt, allowing the user to decrypt a message without directly instantiating a Des3 structure.

Parameters:

- **out** pointer to the buffer in which to store the decrypted plaintext

- **in** pointer to the input buffer containing the encrypted ciphertext
- **sz** length of the ciphertext to decrypt
- **key** pointer to the buffer containing the 24 byte key to use for decryption
- **iv** pointer to the buffer containing the 8 byte iv to use for decryption. If no iv is provided, the iv defaults to 0

See: wc_Des3_CbcDecrypt

Return:

- 0 Returned upon successfully decrypting the given ciphertext
- MEMORY_E Returned if there is an error allocating space for a Des structure

3

Example

```
int ret;
byte key[] = { // initialize with 24 byte key };
byte iv[]  = { // initialize with 8 byte iv };

byte cipher[]  = { // initialize with ciphertext };
byte decoded[sizeof(cipher)];

if ( wc_Des3_CbcDecryptWithKey(decoded, cipher, sizeof(cipher),
key, iv) != 0) {
    // error decrypting message
}
```

C.61.2.6 function wc_AesCbcEncryptWithKey

```
int wc_AesCbcEncryptWithKey(
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * key,
    word32 keySz,
    const byte * iv
)
```

This function encrypts a plaintext message and stores the result in the output buffer. It uses AES encryption with cipher block chaining (CBC) mode. This function does not require an AES structure to be initialized. Instead, it takes in a key and an iv and uses these to encrypt the message.

Parameters:

- **out** pointer to the output buffer in which to store the ciphertext of the encrypted message
- **in** pointer to the input buffer containing plaintext to encrypt
- **inSz** size of input message
- **key** 16, 24, or 32 byte secret key for encryption
- **keySz** size of key used for encryption
- **iv** pointer to the 16 byte initialization vector to use

See:

- wc_AesCbcDecryptWithKey
- wc_AesSetKey
- wc_AesCbcEncrypt

Return:

- 0 On successfully encrypting the message
- BAD_ALIGN_E Returned on block align error
- BAD_FUNC_ARG Returned if key length is invalid
- MEMORY_E Returned if WOLFSSL_SMALL_STACK is enabled and XMALLOC fails to instantiate an AES object

Example

```
byte key[]; // 16, 24, or 32 byte key
byte iv[]; // 16 byte iv
byte plain[]; // plaintext to encrypt
byte cipher[sizeof(plain)];

int ret = wc_AesCbcEncryptWithKey(cipher, plain, sizeof(plain),
                                  key, sizeof(key), iv);
if (ret != 0) {
    // encryption error
}
```

C.61.2.7 function wc_BufferKeyDecrypt

```
int wc_BufferKeyDecrypt(
    struct EncryptedInfo * info,
    byte * der,
    word32 derSz,
    const byte * password,
    int passwordSz,
    int hashType
)
```

This function decrypts an encrypted key buffer using the provided password. It supports encryption algorithms DES, 3DES, and AES. The encryption information is provided in the EncryptedInfo structure.

Parameters:

- **info** pointer to EncryptedInfo structure containing encryption algorithm and parameters
- **der** pointer to the encrypted key buffer
- **derSz** size of the encrypted key buffer
- **password** pointer to the password buffer
- **passwordSz** size of the password
- **hashType** hash algorithm to use for key derivation

See: wc_BufferKeyEncrypt

Return:

- 0 on success
- BAD_FUNC_ARG if der is NULL
- BAD_FUNC_ARG if password is NULL
- BAD_FUNC_ARG if info is NULL
- BAD_FUNC_ARG if info->keySz is 0
- BAD_FUNC_ARG if info->keySz > WC_MAX_SYM_KEY_SIZE
- ALGO_ID_E if info->cipherType is not one of the cipher types this function handles
- BUFFER_E if IV buffer is too small
- NOT_COMPILED_IN if info->cipherType is a cipher type this function handles, but support for it was disabled in this build
- Negative value on error

Example

```
EncryptedInfo info;
byte encryptedKey[32]; // encrypted key data, decrypted in place
byte password[] = "mypassword";

XMEMSET(&info, 0, sizeof(info));
info.cipherType = WC_CIPHER_AES_CBC;
info.keySz = 16;       // size of the key derived from the password

// info.iv holds the IV as hex characters, the way it appears in a PEM
// DEK-Info header.  It is Base16-decoded in place, and must decode to
// at least PKCS5_SALT_SZ bytes.  The decoded bytes are used both as the
// key derivation salt and as the cipher IV.
XMEMCPY(info.iv, "0123456789ABCDEF0123456789ABCDEF", 32);
info.ivSz = 32;        // length of the hex string, not the decoded IV

int ret = wc_BufferKeyDecrypt(&info, encryptedKey,
                              sizeof(encryptedKey), password,
                              sizeof(password)-1, WC_SHA256);
if (ret != 0) {
    // decryption error
}
```

C.61.2.8 function wc_BufferKeyEncrypt

```
int wc_BufferKeyEncrypt(
    struct EncryptedInfo * info,
    byte * der,
    word32 derSz,
    const byte * password,
    int passwordSz,
    int hashType
)
```

This function encrypts a key buffer using the provided password. It supports encryption algorithms DES, 3DES, and AES. The encryption information is provided in the EncryptedInfo structure.

Parameters:

- **info** pointer to EncryptedInfo structure containing encryption algorithm and parameters
- **der** pointer to the key buffer to encrypt
- **derSz** size of the key buffer
- **password** pointer to the password buffer
- **passwordSz** size of the password
- **hashType** hash algorithm to use for key derivation

See: wc_BufferKeyDecrypt

Return:

- 0 on success
- BAD_FUNC_ARG if der is NULL
- BAD_FUNC_ARG if password is NULL
- BAD_FUNC_ARG if info is NULL
- BAD_FUNC_ARG if info->keySz is 0
- BAD_FUNC_ARG if info->keySz > WC_MAX_SYM_KEY_SIZE
- BAD_FUNC_ARG if info->ivSz < PKCS5_SALT_SZ
- ALGO_ID_E if info->cipherType is not one of the cipher types this function handles

- NOT_COMPILED_IN if info->cipherType is a cipher type this function handles, but support for it was disabled in this build
- Negative value on error

Example

```
EncryptedInfo info;
byte key[32]; // key data to encrypt, encrypted in place
byte password[] = "mypassword";

XMEMSET(&info, 0, sizeof(info));
info.cipherType = WC_CIPHER_AES_CBC;
info.keySz = 16;   // size of the key derived from the password
info.ivSz  = 16;   // must be at least PKCS5_SALT_SZ

// Unlike wc_BufferKeyDecrypt, info.iv holds raw bytes here.  Fill it
// with info.ivSz random bytes; the first PKCS5_SALT_SZ of them are used
// as the key derivation salt, and the buffer is also used as the
// cipher IV.
WC_RNG rng;
wc_InitRng(&rng);
wc_RNG_GenerateBlock(&rng, info.iv, info.ivSz);

int ret = wc_BufferKeyEncrypt(&info, key, sizeof(key), password,
                              sizeof(password)-1, WC_SHA256);
if (ret != 0) {
    // encryption error
}
wc_FreeRng(&rng);
```

C.61.3 Source code

```
int  wc_AesCbcDecryptWithKey(byte* out, const byte* in, word32 inSz,
                                        const byte* key, word32 keySz,
                                        const byte* iv);

int  wc_Des_CbcDecryptWithKey(byte* out,
                                         const byte* in, word32 sz,
                                         const byte* key, const byte* iv);

int  wc_Des_CbcEncryptWithKey(byte* out,
                                         const byte* in, word32 sz,
                                         const byte* key, const byte* iv);

int  wc_Des3_CbcEncryptWithKey(byte* out,
                                          const byte* in, word32 sz,
                                          const byte* key, const byte* iv);

int  wc_Des3_CbcDecryptWithKey(byte* out,
                                          const byte* in, word32 sz,
                                          const byte* key, const byte* iv);

int wc_AesCbcEncryptWithKey(byte* out, const byte* in, word32 inSz,
                            const byte* key, word32 keySz,
```

```
                            const byte* iv);

int wc_BufferKeyDecrypt(struct EncryptedInfo* info, byte* der,
                        word32 derSz, const byte* password,
                        int passwordSz, int hashType);

int wc_BufferKeyEncrypt(struct EncryptedInfo* info, byte* der,
                        word32 derSz, const byte* password,
                        int passwordSz, int hashType);
```

C.62 wc_lms.h

C.62.1 Functions

	Name
int	**wc_LmsKey_Init**(LmsKey * key, void * heap, int devId)Initializes an LmsKey object. Must be called before any other LMS/HSS operation. Use wc_LmsKey_Free() to release resources when done.
int	**wc_LmsKey_InitId**(LmsKey * key, const unsigned char * id, int len, void * heap, int devId)Initializes an LmsKey with a device_side key identifier. Equivalent to wc_LmsKey_Init() but also stashes a binary id that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.
int	**wc_LmsKey_InitLabel**(LmsKey * key, const char * label, void * heap, int devId)Initializes an LmsKey with a device_side key label. Equivalent to wc_LmsKey_Init() but also stashes a label string that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.
int	**wc_LmsKey_SetLmsParm**(LmsKey * key, enum wc_LmsParm lmsParm)Selects a predefined LMS/HSS parameter set by name. The enum wc_LmsParm encodes the tree depth (levels), per-tree height, Winternitz parameter and hash family in a single value. See the wc_LmsParm definition for the list of names available in a given build.
int	**wc_LmsKey_SetParameters**(LmsKey * key, int levels, int height, int winternitz)Sets the LMS/HSS parameters individually. The default SHA_256/256 hash is used. For finer control over the hash family use wc_LmsKey_SetParameters_ex().

	Name
int	**wc_LmsKey_SetParameters_ex**(LmsKey * key, int levels, int height, int winternitz, int hash)Sets the LMS/HSS parameters individually with an explicit hash family selector.
int	**wc_LmsKey_GetParameters**(const LmsKey * key, int * levels, int * height, int * winternitz)Retrieves the LMS/HSS parameters previously set on this key.
int	**wc_LmsKey_GetParameters_ex**(const LmsKey * key, int * levels, int * height, int * winternitz, int * hash)Retrieves the LMS/HSS parameters and hash family selector from this key.
int	**wc_LmsKey_SetWriteCb**(LmsKey * key, wc_lms_write_private_key_cb write_cb)Registers the callback that wolfSSL invokes to persist updated private key state. Because LMS/HSS is stateful, the application MUST persist the private key after each successful sign before the signature is released to a peer; otherwise a crash or restart can lead to one-time key reuse and break the scheme.
int	**wc_LmsKey_SetReadCb**(LmsKey * key, wc_lms_read_private_key_cb read_cb)Registers the callback that wolfSSL invokes to load persisted private key state. Used by wc_LmsKey_Reload() to bring a saved key back into memory for further signing.
int	**wc_LmsKey_SetContext**(LmsKey * key, void * context)Sets the opaque context pointer passed to both the read and write private-key callbacks. Typically used to carry a file handle, database connection, or other persistence-layer state.
int	**wc_LmsKey_MakeKey**(LmsKey * key, WC_RNG * rng)Generates a fresh LMS/HSS key pair. Parameters must already be set (via wc_LmsKey_SetLmsParm() or wc_LmsKey_SetParameters()) and read/write callbacks must be registered. The newly generated private key is persisted via the write callback before the function returns; on success the key transitions to state WC_LMS_STATE_OK.
int	**wc_LmsKey_Reload**(LmsKey * key)Reloads a previously generated LMS/HSS private key from persistent storage using the registered read callback, restoring the key to a state where it can sign further messages. On success the key is in state WC_LMS_STATE_OK.
int	**wc_LmsKey_GetPrivLen**(const LmsKey * key, word32 * len)Returns the size in bytes of the encoded private key for the parameter set selected on this key.

	Name
int	**wc_LmsKey_GetPubLen**(const LmsKey * key, word32 * len)Returns the size in bytes of the LMS/HSS public key for the parameter set selected on this key.
int	**wc_LmsKey_GetSigLen**(const LmsKey * key, word32 * len)Returns the LMS/HSS signature size in bytes for the parameter set selected on this key.
int	**wc_LmsKey_Sign**(LmsKey * key, byte * sig, word32 * sigSz, const byte * msg, int msgSz)Signs msg with the LMS/HSS private key in key. On entry sigSz is the size of the sig buffer; on success it is updated to the bytes written.
int	**wc_LmsKey_SigsLeft**(LmsKey * key)Reports whether this key can create another signature. It does not return a count. A key whose one-time keys are used up returns 0 and should be retired.
void	**wc_LmsKey_Free**(LmsKey * key)Releases resources held by an LmsKey. Safe to call with a NULL pointer. After this call the key is in state WC_LMS_STATE_FREED and must be re-initialized before reuse.
int	**wc_LmsKey_ExportPub**(LmsKey * keyDst, const LmsKey * keySrc)Copies the public part of keySrc into keyDst. The destination key inherits the same parameters and may be used for verification; it does not carry the private key state and cannot sign. Useful for handing a verifier the minimal data it needs.
int	**wc_LmsKey_ExportPub_ex**(LmsKey * keyDst, const LmsKey * keySrc, void * heap, int devId)Like wc_LmsKey_ExportPub() but the destination key is initialized fresh with the supplied heap and devId.
int	**wc_LmsKey_ExportPubRaw**(const LmsKey * key, byte * out, word32 * outLen)Exports the LMS/HSS public key as a raw byte string. On entry *outLen is the size of out; on success it is updated to the bytes written.
int	**wc_LmsKey_ImportPubRaw**(LmsKey * key, const byte * in, word32 inLen)Imports a raw LMS/HSS public key into key. The key must be in state WC_LMS_STATE_INITED. Parameter information is recovered from the encoded header, after which the key transitions to state WC_LMS_STATE_VERIFYONLY.

	Name
int	**wc_LmsKey_Verify**(LmsKey * key, const byte * sig, word32 sigSz, const byte * msg, int msgSz)Verifies an LMS/HSS signature against msg using the public key held in key. The function returns 0 only when the signature is valid; any other value indicates the signature was rejected.
const char *	**wc_LmsKey_ParmToStr**(enum wc_LmsParm lmsParm)Returns a static, NUL-terminated string describing an LMS parameter set. Useful for logging and diagnostics.
int	**wc_LmsKey_GetKid**(LmsKey * key, const byte ** kid, word32 * kidSz)Returns a pointer to the 16_byte LMS Key Identifier (I) embedded in the private key, together with its length. The returned pointer aliases internal key memory and is valid only until the key is freed.
const byte *	**wc_LmsKey_GetKidFromPrivRaw**(const byte * priv, word32 privSz)Returns a pointer to the LMS Key Identifier (I) within a raw encoded private key buffer, without requiring an LmsKey object. Used to look up the matching state record in persistent storage during reload.

C.62.2 Functions Documentation

C.62.2.1 function wc_LmsKey_Init

```
int wc_LmsKey_Init(
    LmsKey * key,
    void * heap,
    int devId
)
```

Initializes an LmsKey object. Must be called before any other LMS/HSS operation. Use wc_LmsKey_Free() to release resources when done.

Parameters:

- **key** Pointer to the LmsKey to initialize.
- **heap** Heap hint for dynamic memory allocation. May be NULL.
- **devId** Device identifier for hardware crypto callbacks. Use INVALID_DEVID for software-only.

See:

- wc_LmsKey_Free
- wc_LmsKey_SetLmsParm
- wc_LmsKey_SetParameters
- wc_LmsKey_MakeKey

Return:

- 0 on success.
- BAD_FUNC_ARG if key is NULL.

LMS (Leighton-Micali Signatures) and the HSS multi-tree composition (RFC 8554, NIST SP 800-208) are STATEFUL hash-based signature schemes: each call to wc_LmsKey_Sign() consumes a one_time component of the private key, and reusing a one_time key breaks the security of the scheme. Applications MUST persist the private key state between sign calls; see wc_LmsKey_SetWriteCb() and wc_LmsKey_SetReadCb().

After init the key is in state WC_LMS_STATE_INITED. The parameters must be set with wc_LmsKey_SetLmsParm() or wc_LmsKey_SetParameters() before generating or reloading a key.

Example

```
LmsKey key;
int ret;

ret = wc_LmsKey_Init(&key, NULL, INVALID_DEVID);
if (ret != 0) {
    // error initializing key
}
wc_LmsKey_SetLmsParm(&key, WC_LMS_PARM_L2_H10_W8);
// ... use key ...
wc_LmsKey_Free(&key);
```

C.62.2.2 function wc_LmsKey_InitId

```
int wc_LmsKey_InitId(
    LmsKey * key,
    const unsigned char * id,
    int len,
    void * heap,
    int devId
)
```

Initializes an LmsKey with a device-side key identifier. Equivalent to wc_LmsKey_Init() but also stashes a binary id that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.

Parameters:

- **key** Pointer to the LmsKey to initialize.
- **id** Pointer to the device-side key identifier bytes. May be NULL when len is 0.
- **len** Number of bytes in id; must be in [0, LMS_MAX_ID_LEN].
- **heap** Heap hint for dynamic memory allocation.
- **devId** Device identifier for the crypto callback.

See:

- wc_LmsKey_Init
- wc_LmsKey_InitLabel
- wc_LmsKey_Free

Return:

- 0 on success.
- BAD_FUNC_ARG if key is NULL, or id is NULL while len > 0.
- BUFFER_E if len is negative or greater than LMS_MAX_ID_LEN.

The id is copied into the key object; the caller may free its buffer immediately after this call returns.

C.62.2.3 function wc_LmsKey_InitLabel

```
int wc_LmsKey_InitLabel(
    LmsKey * key,
    const char * label,
    void * heap,
    int devId
)
```

Initializes an LmsKey with a device-side key label. Equivalent to wc_LmsKey_Init() but also stashes a label string that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.

Parameters:

- **key** Pointer to the LmsKey to initialize.
- **label** NUL-terminated device-side key label.
- **heap** Heap hint for dynamic memory allocation.
- **devId** Device identifier for the crypto callback.

See:

- wc_LmsKey_Init
- wc_LmsKey_InitId

Return:

- 0 on success.
- BAD_FUNC_ARG if key or label is NULL.
- BUFFER_E if label is empty or longer than LMS_MAX_LABEL_LEN.

C.62.2.4 function wc_LmsKey_SetLmsParm

```
int wc_LmsKey_SetLmsParm(
    LmsKey * key,
    enum wc_LmsParm lmsParm
)
```

Selects a predefined LMS/HSS parameter set by name. The enum wc_LmsParm encodes the tree depth (levels), per-tree height, Winternitz parameter and hash family in a single value. See the wc_LmsParm definition for the list of names available in a given build.

Parameters:

- **key** Pointer to an LmsKey in state WC_LMS_STATE_INITED.
- **lmsParm** A wc_LmsParm constant (e.g. WC_LMS_PARM_L2_H10_W8).

See:

- wc_LmsKey_SetParameters
- wc_LmsKey_GetParameters
- wc_LmsKey_ParmToStr

Return:

- 0 on success.
- BAD_FUNC_ARG if key is NULL, or if lmsParm is not recognized or names a parameter set that was not compiled in.
- BAD_STATE_E if key is not in state WC_LMS_STATE_INITED.

Example

```
LmsKey key;

wc_LmsKey_Init(&key, NULL, INVALID_DEVID);
wc_LmsKey_SetLmsParm(&key, WC_LMS_PARM_L2_H10_W8);
```

C.62.2.5 function wc_LmsKey_SetParameters

```
int wc_LmsKey_SetParameters(
    LmsKey * key,
    int levels,
    int height,
    int winternitz
)
```

Sets the LMS/HSS parameters individually. The default SHA-256/256 hash is used. For finer control over the hash family use wc_LmsKey_SetParameters_ex().

Parameters:

- **key** Pointer to an LmsKey in state WC_LMS_STATE_INITED.
- **levels** Number of Merkle-tree levels in the HSS chain.
- **height** Height of each individual Merkle tree.
- **winternitz** Winternitz parameter (1, 2, 4 or 8).

See:

- wc_LmsKey_SetParameters_ex
- wc_LmsKey_SetLmsParm
- wc_LmsKey_GetParameters

Return:

- 0 on success.
- BAD_FUNC_ARG if key is NULL, or if the requested parameter combination is not supported in this build.
- BAD_STATE_E if key is not in state WC_LMS_STATE_INITED.

The combination of parameters must match one of the allowed sets in RFC 8554:

- levels: 1..8
- height: 5, 10, 15, 20 (and 25 in some builds)
- winternitz: 1, 2, 4, or 8

The maximum number of signatures available from a key is 2^(levels * height).

C.62.2.6 function wc_LmsKey_SetParameters_ex

```
int wc_LmsKey_SetParameters_ex(
    LmsKey * key,
    int levels,
    int height,
    int winternitz,
    int hash
)
```

Sets the LMS/HSS parameters individually with an explicit hash family selector.

Parameters:

- **key** Pointer to an LmsKey in state WC_LMS_STATE_INITED.
- **levels** Number of Merkle-tree levels.

- **height** Height of each tree.
- **winternitz** Winternitz parameter (1, 2, 4 or 8).
- **hash** Hash family selector identifying SHA-256/256, SHA-256/192, SHAKE256/256 or SHAKE256/192, as supported by the build.

See:

- wc_LmsKey_SetParameters
- wc_LmsKey_GetParameters_ex

Return:

- 0 on success.
- BAD_FUNC_ARG if key is NULL, or if the requested parameter combination is not supported in this build.
- BAD_STATE_E if key is not in state WC_LMS_STATE_INITED.

C.62.2.7 function wc_LmsKey_GetParameters

```
int wc_LmsKey_GetParameters(
    const LmsKey * key,
    int * levels,
    int * height,
    int * winternitz
)
```

Retrieves the LMS/HSS parameters previously set on this key.

Parameters:

- **key** Pointer to an LmsKey with parameters set.
- **levels** Receives the number of tree levels.
- **height** Receives the per-tree height.
- **winternitz** Receives the Winternitz parameter.

See:

- wc_LmsKey_SetParameters
- wc_LmsKey_GetParameters_ex

Return:

- 0 on success.
- BAD_FUNC_ARG if any pointer is NULL or no parameters are set.

C.62.2.8 function wc_LmsKey_GetParameters_ex

```
int wc_LmsKey_GetParameters_ex(
    const LmsKey * key,
    int * levels,
    int * height,
    int * winternitz,
    int * hash
)
```

Retrieves the LMS/HSS parameters and hash family selector from this key.

Parameters:

- **key** Pointer to an LmsKey with parameters set.
- **levels** Receives the number of tree levels.

- **height** Receives the per-tree height.
- **winternitz** Receives the Winternitz parameter.
- **hash** Receives the hash family selector.

See: wc_LmsKey_SetParameters_ex

Return:

- 0 on success.
- BAD_FUNC_ARG if any pointer is NULL or no parameters are set.

C.62.2.9 function wc_LmsKey_SetWriteCb

```
int wc_LmsKey_SetWriteCb(
    LmsKey * key,
    wc_lms_write_private_key_cb write_cb
)
```

Registers the callback that wolfSSL invokes to persist updated private key state. Because LMS/HSS is stateful, the application MUST persist the private key after each successful sign before the signature is released to a peer; otherwise a crash or restart can lead to one-time key reuse and break the scheme.

Parameters:

- **key** Pointer to an LmsKey.
- **write_cb** Callback invoked to persist the private key.

See:

- wc_LmsKey_SetReadCb
- wc_LmsKey_SetContext
- wc_LmsKey_Sign

Return:

- 0 on success.
- BAD_FUNC_ARG if key or write_cb is NULL.

The callback receives the encoded private key bytes and returns one of the wc_LmsRc codes. WC_LMS_RC_SAVED_TO_NV_MEMORY signals a durable write; other return codes are treated as failures.

C.62.2.10 function wc_LmsKey_SetReadCb

```
int wc_LmsKey_SetReadCb(
    LmsKey * key,
    wc_lms_read_private_key_cb read_cb
)
```

Registers the callback that wolfSSL invokes to load persisted private key state. Used by wc_LmsKey_Reload() to bring a saved key back into memory for further signing.

Parameters:

- **key** Pointer to an LmsKey.
- **read_cb** Callback invoked to load the private key.

See:

- wc_LmsKey_SetWriteCb
- wc_LmsKey_SetContext
- wc_LmsKey_Reload

Return:

- 0 on success.
- BAD_FUNC_ARG if key or read_cb is NULL.

C.62.2.11 function wc_LmsKey_SetContext

```
int wc_LmsKey_SetContext(
    LmsKey * key,
    void * context
)
```

Sets the opaque context pointer passed to both the read and write private-key callbacks. Typically used to carry a file handle, database connection, or other persistence-layer state.

Parameters:

- **key** Pointer to an LmsKey.
- **context** Application-defined pointer; may be NULL.

See:

- wc_LmsKey_SetReadCb
- wc_LmsKey_SetWriteCb

Return:

- 0 on success.
- BAD_FUNC_ARG if key is NULL.

C.62.2.12 function wc_LmsKey_MakeKey

```
int wc_LmsKey_MakeKey(
    LmsKey * key,
    WC_RNG * rng
)
```

Generates a fresh LMS/HSS key pair. Parameters must already be set (via wc_LmsKey_SetLmsParm() or wc_LmsKey_SetParameters()) and read/write callbacks must be registered. The newly generated private key is persisted via the write callback before the function returns; on success the key transitions to state WC_LMS_STATE_OK.

Parameters:

- **key** Pointer to an LmsKey in state WC_LMS_STATE_PARMSET with callbacks set.
- **rng** Pointer to an initialized WC_RNG.

See:

- wc_LmsKey_Sign
- wc_LmsKey_Reload

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- MEMORY_E on allocation failure.

Key generation runtime grows quickly with the first tree's height: a 3-level h=5 tree is much faster than a 1-level h=15 tree even though both yield the same total signature count.

Example

```
LmsKey key;
WC_RNG rng;

wc_LmsKey_Init(&key, NULL, INVALID_DEVID);
wc_LmsKey_SetLmsParm(&key, WC_LMS_PARM_L2_H10_W8);
wc_LmsKey_SetWriteCb(&key, my_write_cb);
wc_LmsKey_SetReadCb(&key, my_read_cb);
wc_LmsKey_SetContext(&key, &my_storage);
wc_InitRng(&rng);

if (wc_LmsKey_MakeKey(&key, &rng) != 0) {
    // error generating key
}
```

C.62.2.13 function wc_LmsKey_Reload

```
int wc_LmsKey_Reload(
    LmsKey * key
)
```

Reloads a previously generated LMS/HSS private key from persistent storage using the registered read callback, restoring the key to a state where it can sign further messages. On success the key is in state WC_LMS_STATE_OK.

Parameters:

- **key** Pointer to an LmsKey with parameters and read callback set.

See:

- wc_LmsKey_MakeKey
- wc_LmsKey_SetReadCb

Return:

- 0 on success, including the device-backed no-op.
- BAD_FUNC_ARG if any required pointer is NULL.
- WC_LMS_RC_* mapped error if the read callback fails.
- IO_FAILED_E if the private key could not be read.
- NO_VALID_DEVID with WOLF_CRYPTO_CB_ONLY_LMS when the key is not device-backed.

The same parameters set at key-generation time must be reapplied to the LmsKey before calling Reload (the persisted blob is just the private key bytes; the parameter set is metadata the application owns).

With WOLF_CRYPTO_CB, a key on a devId with no read callback is taken to be device-backed and Reload is a no-op. With a read callback the software reload runs as usual.

WOLF_CRYPTO_CB_ONLY_LMS removes the software reload, so the read callback is never called and only the device-backed no-op succeeds.

A reloaded key holds no public key: neither arm populates it. Call wc_LmsKey_ImportPubRaw() on a separate key to verify, or keep the public key exported at generation time.

C.62.2.14 function wc_LmsKey_GetPrivLen

```
int wc_LmsKey_GetPrivLen(
    const LmsKey * key,
    word32 * len
)
```

Returns the size in bytes of the encoded private key for the parameter set selected on this key.

Parameters:

- **key** Pointer to an LmsKey with parameters set.
- **len** Receives the private key size in bytes.

See:

- wc_LmsKey_GetPubLen
- wc_LmsKey_GetSigLen

Return:

- 0 on success.
- BAD_FUNC_ARG if key or len is NULL.

C.62.2.15 function wc_LmsKey_GetPubLen

```
int wc_LmsKey_GetPubLen(
    const LmsKey * key,
    word32 * len
)
```

Returns the size in bytes of the LMS/HSS public key for the parameter set selected on this key.

Parameters:

- **key** Pointer to an LmsKey with parameters set.
- **len** Receives the public key size in bytes.

See:

- wc_LmsKey_ExportPubRaw
- wc_LmsKey_GetPrivLen

Return:

- 0 on success.
- BAD_FUNC_ARG if key or len is NULL.

C.62.2.16 function wc_LmsKey_GetSigLen

```
int wc_LmsKey_GetSigLen(
    const LmsKey * key,
    word32 * len
)
```

Returns the LMS/HSS signature size in bytes for the parameter set selected on this key.

Parameters:

- **key** Pointer to an LmsKey with parameters set.
- **len** Receives the signature size in bytes.

See: wc_LmsKey_Sign

Return:

- 0 on success.
- BAD_FUNC_ARG if key or len is NULL.

C.62.2.17 function wc_LmsKey_Sign

```
int wc_LmsKey_Sign(
    LmsKey * key,
    byte * sig,
    word32 * sigSz,
    const byte * msg,
    int msgSz
)
```

Signs msg with the LMS/HSS private key in key. On entry sigSz is the size of the sig buffer; on success it is updated to the bytes written.

Parameters:

- **key** Pointer to an LmsKey in state WC_LMS_STATE_OK.
- **sig** Buffer that receives the signature.
- **sigSz** In: size of sig. Out: bytes written.
- **msg** Message to sign.
- **msgSz** Length of msg in bytes.

See:

- wc_LmsKey_Verify
- wc_LmsKey_SigsLeft
- wc_LmsKey_SetWriteCb

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- BUFFER_E if *sigSz is smaller than the signature size.
- -1 (or similar) if all one-time keys have been used.

Each successful sign call consumes a one-time component of the private key. The updated key state is persisted via the registered write callback BEFORE the new signature is returned to the caller. If the write callback fails the sign call fails and the signature is not released. When the supply of one-time keys is exhausted the key transitions to state WC_LMS_STATE_NOSIGS and further sign attempts return SIG_OTHER_E (or similar) – query wc_LmsKey_SigsLeft() to detect this condition in advance.

C.62.2.18 function wc_LmsKey_SigsLeft

```
int wc_LmsKey_SigsLeft(
    LmsKey * key
)
```

Reports whether this key can create another signature. It does not return a count. A key whose one-time keys are used up returns 0 and should be retired.

Parameters:

- **key** Pointer to an LmsKey in state WC_LMS_STATE_OK.

See: wc_LmsKey_Sign

Return:

- 1 if at least one signature remains.
- 0 if the one-time keys are exhausted, if key is NULL, has no parameters set or is verify-only, or if a crypto callback device returns an error.

C.62.2.19 function wc_LmsKey_Free

```
void wc_LmsKey_Free(
    LmsKey * key
)
```

Releases resources held by an LmsKey. Safe to call with a NULL pointer. After this call the key is in state WC_LMS_STATE_FREED and must be re-initialized before reuse.

Parameters:

- **key** Pointer to the LmsKey to free.

See: wc_LmsKey_Init

C.62.2.20 function wc_LmsKey_ExportPub

```
int wc_LmsKey_ExportPub(
    LmsKey * keyDst,
    const LmsKey * keySrc
)
```

Copies the public part of keySrc into keyDst. The destination key inherits the same parameters and may be used for verification; it does not carry the private key state and cannot sign. Useful for handing a verifier the minimal data it needs.

Parameters:

- **keyDst** Pointer to an initialized destination LmsKey.
- **keySrc** Pointer to an LmsKey with the public key.

See:

- wc_LmsKey_ExportPub_ex
- wc_LmsKey_ExportPubRaw

Return:

- 0 on success.
- BAD_FUNC_ARG if keyDst or keySrc is NULL.
- BAD_STATE_E if keySrc holds no public key.

C.62.2.21 function wc_LmsKey_ExportPub_ex

```
int wc_LmsKey_ExportPub_ex(
    LmsKey * keyDst,
    const LmsKey * keySrc,
    void * heap,
    int devId
)
```

Like wc_LmsKey_ExportPub() but the destination key is initialized fresh with the supplied heap and devId.

Parameters:

- **keyDst** Pointer to an LmsKey to populate.
- **keySrc** Pointer to an LmsKey with the public key.
- **heap** Heap hint for keyDst.
- **devId** Device identifier for keyDst.

See: wc_LmsKey_ExportPub

Return:

- 0 on success.
- BAD_FUNC_ARG if keyDst or keySrc is NULL.
- BAD_STATE_E if keySrc holds no public key.

C.62.2.22 function wc_LmsKey_ExportPubRaw

```
int wc_LmsKey_ExportPubRaw(
    const LmsKey * key,
    byte * out,
    word32 * outLen
)
```

Exports the LMS/HSS public key as a raw byte string. On entry *outLen is the size of out; on success it is updated to the bytes written.

Parameters:

- **key** Pointer to an LmsKey.
- **out** Buffer that receives the public key.
- **outLen** In: size of out. Out: bytes written.

See:

- wc_LmsKey_ImportPubRaw
- wc_LmsKey_GetPubLen

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- BAD_STATE_E if key holds no public key.
- BUFFER_E if *outLen is smaller than the public key size.

C.62.2.23 function wc_LmsKey_ImportPubRaw

```
int wc_LmsKey_ImportPubRaw(
    LmsKey * key,
    const byte * in,
    word32 inLen
)
```

Imports a raw LMS/HSS public key into key. The key must be in state WC_LMS_STATE_INITED. Parameter information is recovered from the encoded header, after which the key transitions to state WC_LMS_STATE_VERIFYONLY.

Parameters:

- **key** Pointer to an LmsKey in WC_LMS_STATE_INITED.
- **in** Raw public key bytes.
- **inLen** Length of in in bytes.

See:

- wc_LmsKey_ExportPubRaw
- wc_LmsKey_Verify

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- BUFFER_E if inLen is too small.

C.62.2.24 function wc_LmsKey_Verify

```
int wc_LmsKey_Verify(
    LmsKey * key,
    const byte * sig,
    word32 sigSz,
    const byte * msg,
    int msgSz
)
```

Verifies an LMS/HSS signature against msg using the public key held in key. The function returns 0 only when the signature is valid; any other value indicates the signature was rejected.

Parameters:

- **key** Pointer to an LmsKey with the public key set.
- **sig** Signature bytes to verify.
- **sigSz** Length of sig in bytes.
- **msg** Message that was signed.
- **msgSz** Length of msg in bytes.

See:

- wc_LmsKey_Sign
- wc_LmsKey_ImportPubRaw

Return:

- 0 on a valid signature.
- BAD_FUNC_ARG if any required pointer is NULL.
- BAD_STATE_E if key holds no public key.
- SIG_VERIFY_E (or similar) if the signature is invalid or malformed.

C.62.2.25 function wc_LmsKey_ParmToStr

```
const char * wc_LmsKey_ParmToStr(
    enum wc_LmsParm lmsParm
)
```

Returns a static, NUL-terminated string describing an LMS parameter set. Useful for logging and diagnostics.

Parameters:

- **lmsParm** A wc_LmsParm constant.

See: wc_LmsKey_SetLmsParm

Return:

- Pointer to a static string on success.
- NULL if lmsParm is not recognized.

C.62.2.26 function wc_LmsKey_GetKid

```
int wc_LmsKey_GetKid(
    LmsKey * key,
    const byte ** kid,
    word32 * kidSz
)
```

Returns a pointer to the 16-byte LMS Key Identifier (I) embedded in the private key, together with its length. The returned pointer aliases internal key memory and is valid only until the key is freed.

Parameters:

- **key** Pointer to an LmsKey with a private key.
- **kid** Receives a pointer to the I bytes.
- **kidSz** Receives the length (16 / WC_LMS_I_LEN).

See: wc_LmsKey_GetKidFromPrivRaw

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- BAD_STATE_E if the private key is not in memory, such as when a device keeps it.

C.62.2.27 function wc_LmsKey_GetKidFromPrivRaw

```
const byte * wc_LmsKey_GetKidFromPrivRaw(
    const byte * priv,
    word32 privSz
)
```

Returns a pointer to the LMS Key Identifier (I) within a raw encoded private key buffer, without requiring an LmsKey object. Used to look up the matching state record in persistent storage during reload.

Parameters:

- **priv** Encoded private key bytes.
- **privSz** Length of priv in bytes.

See: wc_LmsKey_GetKid

Return:

- Pointer to the I bytes within priv on success.
- NULL if priv is NULL or privSz is too small to contain a valid header.

C.62.3 Source code

```
int wc_LmsKey_Init(LmsKey* key, void* heap, int devId);

int wc_LmsKey_InitId(LmsKey* key, const unsigned char* id, int len,
    void* heap, int devId);

int wc_LmsKey_InitLabel(LmsKey* key, const char* label, void* heap,
    int devId);

int wc_LmsKey_SetLmsParm(LmsKey* key, enum wc_LmsParm lmsParm);

int wc_LmsKey_SetParameters(LmsKey* key, int levels, int height,
    int winternitz);
```

```
int wc_LmsKey_SetParameters_ex(LmsKey* key, int levels, int height,
    int winternitz, int hash);

int wc_LmsKey_GetParameters(const LmsKey* key, int* levels, int* height,
    int* winternitz);

int wc_LmsKey_GetParameters_ex(const LmsKey* key, int* levels, int* height,
    int* winternitz, int* hash);

int wc_LmsKey_SetWriteCb(LmsKey* key, wc_lms_write_private_key_cb write_cb);

int wc_LmsKey_SetReadCb(LmsKey* key, wc_lms_read_private_key_cb read_cb);

int wc_LmsKey_SetContext(LmsKey* key, void* context);

int wc_LmsKey_MakeKey(LmsKey* key, WC_RNG* rng);

int wc_LmsKey_Reload(LmsKey* key);

int wc_LmsKey_GetPrivLen(const LmsKey* key, word32* len);

int wc_LmsKey_GetPubLen(const LmsKey* key, word32* len);

int wc_LmsKey_GetSigLen(const LmsKey* key, word32* len);

int wc_LmsKey_Sign(LmsKey* key, byte* sig, word32* sigSz, const byte* msg,
    int msgSz);

int wc_LmsKey_SigsLeft(LmsKey* key);

void wc_LmsKey_Free(LmsKey* key);

int wc_LmsKey_ExportPub(LmsKey* keyDst, const LmsKey* keySrc);

int wc_LmsKey_ExportPub_ex(LmsKey* keyDst, const LmsKey* keySrc, void* heap,
    int devId);

int wc_LmsKey_ExportPubRaw(const LmsKey* key, byte* out, word32* outLen);

int wc_LmsKey_ImportPubRaw(LmsKey* key, const byte* in, word32 inLen);

int wc_LmsKey_Verify(LmsKey* key, const byte* sig, word32 sigSz,
    const byte* msg, int msgSz);

const char* wc_LmsKey_ParmToStr(enum wc_LmsParm lmsParm);

int wc_LmsKey_GetKid(LmsKey* key, const byte** kid, word32* kidSz);

const byte* wc_LmsKey_GetKidFromPrivRaw(const byte* priv, word32 privSz);
```

COPYRIGHT ©2024 wolfSSL Inc.

C.63 wc_mldsa.h

C.63.1 Functions

	Name
int	**wc_MlDsaKey_Init**(wc_MlDsaKey * key, void * heap, int devId)Initializes a wc_MlDsaKey object. Must be called before any other ML_DSA operation. Use wc_MlDsaKey_Free() to release resources when done.
wc_MlDsaKey *	**wc_MlDsaKey_New**(void * heap, int devId)Allocates and initializes a new wc_MlDsaKey on the heap. The returned pointer must be released with wc_MlDsaKey_Delete(). Only available when wolfSSL is built without WC_NO_CONSTRUCTORS.
int	**wc_MlDsaKey_Delete**(wc_MlDsaKey * key, wc_MlDsaKey ** key_p)Frees and zeros a heap_allocated wc_MlDsaKey previously returned by wc_MlDsaKey_New(). On success the caller's pointer variable is set to NULL via key_p when key_p is not NULL. Only available when wolfSSL is built without WC_NO_CONSTRUCTORS.
int	**wc_MlDsaKey_InitId**(wc_MlDsaKey * key, const unsigned char * id, int len, void * heap, int devId)Initializes a wc_MlDsaKey with a device_side key identifier. Equivalent to wc_MlDsaKey_Init() but also stashes a binary id that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.
int	**wc_MlDsaKey_InitLabel**(wc_MlDsaKey * key, const char * label, void * heap, int devId)Initializes a wc_MlDsaKey with a device_side key label. Equivalent to wc_MlDsaKey_Init() but also stashes a label string that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.
int	**wc_MlDsaKey_SetParams**(wc_MlDsaKey * key, byte level)Selects the ML_DSA parameter set for this key. Must be called after wc_MlDsaKey_Init() and before key generation, signing or verifying.
int	**wc_MlDsaKey_GetParams**(wc_MlDsaKey * key, byte * level)Retrieves the ML-DSA parameter set currently selected on this key.

	Name
void	**wc_MlDsaKey_Free**(wc_MlDsaKey * key)Releases resources held by a wc_MlDsaKey. After this call the object must be re_initialized with wc_MlDsaKey_Init() before it can be used again. Safe to call with a NULL pointer.
int	**wc_MlDsaKey_MakeKey**(wc_MlDsaKey * key, WC_RNG * rng)Generates a new ML_DSA key pair using the provided RNG. The parameter set must already be set with wc_MlDsaKey_SetParams(). Both the public and private key components are populated on success.
int	**wc_MlDsaKey_MakeKeyFromSeed**(wc_MlDsaKey * key, const byte * seed)Deterministically generates an ML-DSA key pair from a 32-byte seed. Useful for known-answer tests and for applications that derive the seed from another secret. The seed buffer must contain exactly 32 bytes.
int	**wc_MlDsaKey_MakePublicKey**(wc_MlDsaKey * key)Derives public key for a wc_MlDsaKey with private key set. If the public key is already set, it is only checked against the private key's tr (H(pk)); use wc_MlDsaKey_CheckKey() for a full key pair check. The private key is still required: a public-only key returns BAD_FUNC_ARG. Available when WC_MLDSA_HAVE_MAKE_PUBLIC_KEY is defined: that is, when WOLFSSL_MLDSA_ASSIGN_KEY, WOLFSSL_MLDSA_NO_MAKE_KEY and WOLFSSL_MLDSA_VERIFY_ONLY are not defined.
int	**wc_MlDsaKey_SignCtx**(wc_MlDsaKey * key, const byte * ctx, byte ctxLen, byte * sig, word32 * sigLen, const byte * msg, word32 msgLen, WC_RNG * rng)Signs a message with ML-DSA using the FIPS 204 randomized-with-context signing API. Pass ctx=NULL and ctxLen=0 for an empty context.
int	**wc_MlDsaKey_SignCtxHash**(wc_MlDsaKey * key, const byte * ctx, byte ctxLen, byte * sig, word32 * sigLen, const byte * hash, word32 hashLen, int hashAlg, WC_RNG * rng)HashML-DSA signing variant: signs a pre-hashed message. The caller supplies the hash bytes and identifies the hash algorithm. This is the "pre-hash" mode of FIPS 204.

	Name
int	**wc_MlDsaKey_Sign**(wc_MlDsaKey * key, byte * sig, word32 * sigLen, const byte * msg, word32 msgLen, WC_RNG * rng)Legacy ML_DSA signing API without a context parameter. Only available when wolfSSL is built with WOLFSSL_MLDSA_NO_CTX. New code should call wc_MlDsaKey_SignCtx() with ctx=NULL and ctxLen=0 for a FIPS 204 compliant empty-context signature.
int	**wc_MlDsaKey_SignCtxWithSeed**(wc_MlDsaKey * key, const byte * ctx, byte ctxLen, byte * sig, word32 * sigLen, const byte * msg, word32 msgLen, const byte * seed)Deterministic signing variant of wc_MlDsaKey_SignCtx(). The 32-byte seed replaces the randomness an RNG would supply, so the same key/ctx/msg/seed always produces the same signature.
int	**wc_MlDsaKey_SignCtxHashWithSeed**(wc_MlDsaKey * key, const byte * ctx, byte ctxLen, byte * sig, word32 * sigLen, const byte * hash, word32 hashLen, int hashAlg, const byte * seed)Deterministic HashML_DSA signing: like wc_MlDsaKey_SignCtxHash() but uses the supplied 32-byte seed in place of an RNG.
int	**wc_MlDsaKey_SignMuWithSeed**(wc_MlDsaKey * key, byte * sig, word32 * sigLen, const byte * mu, word32 muLen, const byte * seed)Signs a pre_computed mu value (the externally derived SHAKE256 hash of (tr
int	**wc_MlDsaKey_SignWithSeed**(wc_MlDsaKey * key, byte * sig, word32 * sigLen, const byte * msg, word32 msgLen, const byte * seed)Legacy seed_based signing API without a context parameter. Only available when wolfSSL is built with WOLFSSL_MLDSA_NO_CTX. New code should use wc_MlDsaKey_SignCtxWithSeed().
int	**wc_MlDsaKey_VerifyCtx**(wc_MlDsaKey * key, const byte * sig, word32 sigLen, const byte * ctx, byte ctxLen, const byte * msg, word32 msgLen, int * res)Verifies an ML_DSA signature produced by wc_MlDsaKey_SignCtx() or one of its variants. On entry res is set to 0; on success it is set to 1 when the signature is valid and left at 0 otherwise. The function's return value reports whether verification could be carried out at all; a bad signature is NOT a function-level error.

	Name
int	**wc_MlDsaKey_VerifyCtxHash**(wc_MlDsaKey * key, const byte * sig, word32 sigLen, const byte * ctx, byte ctxLen, const byte * hash, word32 hashLen, int hashAlg, int * res)Verifies a HashML_DSA signature where the message digest was supplied directly. See wc_MlDsaKey_VerifyCtx() for the semantics of res.
int	**wc_MlDsaKey_VerifyMu**(wc_MlDsaKey * key, const byte * sig, word32 sigLen, const byte * mu, word32 muLen, int * res)Verifies a signature over a pre_computed 64_byte mu value (see wc_MlDsaKey_SignMuWithSeed()). See wc_MlDsaKey_VerifyCtx() for the semantics of res.
int	**wc_MlDsaKey_SetPrecompA**(wc_MlDsaKey * key, const sword32 * a, word32 aLen, const byte * rho, word32 rhoLen)Attaches a matrix A that was expanded off target, so verification can skip the SHAKE128 rejection sampling that otherwise dominates it. A is a function of the public seed rho alone, so wherever the verification key is fixed at build time - secure boot being the usual case - A can be computed once on a host and stored in flash. Available only when WOLFSSL_MLDSA_VERIFY_PRECOMP_A is defined, and honoured by both the default and the small-memory verifiers.
int	**wc_MlDsaKey_Verify**(wc_MlDsaKey * key, const byte * sig, word32 sigLen, const byte * msg, word32 msgLen, int * res)Legacy ML_DSA verify API without a context parameter. Only available when wolfSSL is built with WOLFSSL_MLDSA_NO_CTX. New code should use wc_MlDsaKey_VerifyCtx() with ctx=NULL and ctxLen=0.
int	**wc_MlDsaKey_Size**(wc_MlDsaKey * key)Returns the size in bytes of the encoded private key for the parameter set selected on this key. Equivalent to wc_MlDsaKey_PrivSize() and provided for API compatibility.
int	**wc_MlDsaKey_PrivSize**(wc_MlDsaKey * key)Returns the size in bytes of the encoded private key for the parameter set selected on this key.
int	**wc_MlDsaKey_PubSize**(wc_MlDsaKey * key)Returns the size in bytes of the encoded public key for the parameter set selected on this key.

	Name
int	**wc_MlDsaKey_SigSize**(wc_MlDsaKey * key)Returns the size in bytes of the signature produced by this key's parameter set.
int	**wc_MlDsaKey_GetPrivLen**(wc_MlDsaKey * key, int * len)Writes the encoded private key size into *len. Equivalent information to wc_MlDsaKey_PrivSize() but uses an out-parameter style.
int	**wc_MlDsaKey_GetPubLen**(wc_MlDsaKey * key, int * len)Writes the encoded public key size into *len. Equivalent information to wc_MlDsaKey_PubSize() but uses an out-parameter style.
int	**wc_MlDsaKey_GetSigLen**(wc_MlDsaKey * key, int * len)Writes the signature size into *len. Equivalent information to wc_MlDsaKey_SigSize() but uses an out-parameter style.
int	**wc_MlDsaKey_CheckKey**(wc_MlDsaKey * key)Self-checks an ML-DSA key by recomputing the public key from the private and comparing against the stored public. Only available when wolfSSL is built with WOLFSSL_MLDSA_CHECK_KEY.
int	**wc_MlDsaKey_ImportPubRaw**(wc_MlDsaKey * key, const byte * in, word32 inLen)Imports a raw ML_DSA public key. The parameter set must already be selected on the key. inLen must match the size returned by wc_MlDsaKey_PubSize() for the selected parameter set.
int	**wc_MlDsaKey_ImportPrivRaw**(wc_MlDsaKey * key, const byte * priv, word32 privSz)Imports a raw ML_DSA private key. The parameter set must already be selected. privSz must match the size returned by wc_MlDsaKey_PrivSize().
int	**wc_MlDsaKey_ImportKey**(wc_MlDsaKey * key, const byte * priv, word32 privSz, const byte * pub, word32 pubSz)Imports a raw ML_DSA key pair (private and public parts together). The parameter set must already be selected.
int	**wc_MlDsaKey_ExportPubRaw**(wc_MlDsaKey * key, byte * out, word32 * outLen)Exports the raw ML-DSA public key. On entry *outLen is the size of out; on success it is updated to the bytes written.
int	**wc_MlDsaKey_ExportPrivRaw**(wc_MlDsaKey * key, byte * out, word32 * outLen)Exports the raw ML-DSA private key. On entry *outLen is the size of out; on success it is updated to the bytes written.

	Name
int	**wc_MlDsaKey_ExportKey**(wc_MlDsaKey * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz)Exports both raw public and private ML-DSA key components in one call.
int	**wc_MlDsaKey_PrivateKeyDecode**(wc_MlDsaKey * key, const byte * input, word32 inSz, word32 * inOutIdx)Parses an ML_DSA private key from a DER/ASN.1 encoded buffer (PKCS#8 OneAsymmetricKey). The parameter set is inferred from the algorithm identifier in the encoding, so it does NOT need to be set beforehand. On success *inOutIdx is advanced past the consumed bytes.
int	**wc_MlDsaKey_PublicKeyDecode**(wc_MlDsaKey * key, const byte * input, word32 inSz, word32 * inOutIdx)Parses an ML_DSA public key from a DER/ASN.1 encoded buffer (SubjectPublicKeyInfo). The parameter set is inferred from the algorithm identifier. On success *inOutIdx is advanced past the consumed bytes.
int	**wc_MlDsaKey_PublicKeyToDer**(wc_MlDsaKey * key, byte * output, word32 inLen, int withAlg)Encodes an ML_DSA public key to DER. When withAlg is non_zero the output is a full SubjectPublicKeyInfo (with AlgorithmIdentifier); when zero the output is the raw public key bytes.
int	**wc_MlDsaKey_KeyToDer**(wc_MlDsaKey * key, byte * output, word32 inLen)Encodes an ML_DSA key pair (public + private) to DER as a PKCS#8 OneAsymmetricKey structure. Pass NULL as output to query the required buffer size.
int	**wc_MlDsaKey_PrivateKeyToDer**(wc_MlDsaKey * key, byte * output, word32 inLen)Encodes the ML_DSA private key to DER. Per FIPS 204 the private key encoding includes the public component, so this function is currently an alias of wc_MlDsaKey_KeyToDer() kept for API parity with other algorithms.

C.63.2 Functions Documentation

C.63.2.1 function wc_MlDsaKey_Init

```
int wc_MlDsaKey_Init(
    wc_MlDsaKey * key,
    void * heap,
    int devId
)
```

Initializes a wc_MlDsaKey object. Must be called before any other ML-DSA operation. Use

wc_MlDsaKey_Free() to release resources when done.

Parameters:

- **key** Pointer to the wc_MlDsaKey to initialize.
- **heap** Heap hint for dynamic memory allocation. May be NULL.
- **devId** Device identifier for hardware crypto callbacks. Use INVALID_DEVID for software-only.

See:

- wc_MlDsaKey_Free
- wc_MlDsaKey_SetParams
- wc_MlDsaKey_MakeKey

Return:

- 0 on success.
- BAD_FUNC_ARG if key is NULL.
- Other negative value if initializing the key's SHAKE-256 object fails.

ML-DSA (FIPS 204) is a quantum-resistant digital signature algorithm. Three parameter sets are defined and selected via wc_MlDsaKey_SetParams() after init:

- WC_ML_DSA_44 (NIST security level 2),
- WC_ML_DSA_65 (level 3),
- WC_ML_DSA_87 (level 5).

Example

```
wc_MlDsaKey key;
int ret;

ret = wc_MlDsaKey_Init(&key, NULL, INVALID_DEVID);
if (ret != 0) {
    // error initializing key
}
ret = wc_MlDsaKey_SetParams(&key, WC_ML_DSA_65);
// ... use key ...
wc_MlDsaKey_Free(&key);
```

C.63.2.2 function wc_MlDsaKey_New

```
wc_MlDsaKey * wc_MlDsaKey_New(
    void * heap,
    int devId
)
```

Allocates and initializes a new wc_MlDsaKey on the heap. The returned pointer must be released with wc_MlDsaKey_Delete(). Only available when wolfSSL is built without WC_NO_CONSTRUCTORS.

Parameters:

- **heap** Heap hint for dynamic memory allocation. May be NULL.
- **devId** Device identifier for hardware crypto callbacks. Use INVALID_DEVID for software-only.

See:

- wc_MlDsaKey_Delete
- wc_MlDsaKey_Init

Return:

- Pointer to a freshly allocated wc_MlDsaKey on success.

- NULL on allocation failure.

Example

```
wc_MlDsaKey* key = wc_MlDsaKey_New(NULL, INVALID_DEVID);
if (key == NULL) {
    // allocation failed
}
// ... use key ...
wc_MlDsaKey_Delete(key, &key);
```

C.63.2.3 function wc_MlDsaKey_Delete

```
int wc_MlDsaKey_Delete(
    wc_MlDsaKey * key,
    wc_MlDsaKey ** key_p
)
```

Frees and zeros a heap-allocated wc_MlDsaKey previously returned by wc_MlDsaKey_New(). On success the caller's pointer variable is set to NULL via key_p when key_p is not NULL. Only available when wolfSSL is built without WC_NO_CONSTRUCTORS.

Parameters:

- **key** The wc_MlDsaKey to free.
- **key_p** Optional address of the caller's pointer variable; when not NULL, it is set to NULL on success.

See: wc_MlDsaKey_New

Return:

- 0 on success.
- BAD_FUNC_ARG if key is NULL.

C.63.2.4 function wc_MlDsaKey_InitId

```
int wc_MlDsaKey_InitId(
    wc_MlDsaKey * key,
    const unsigned char * id,
    int len,
    void * heap,
    int devId
)
```

Initializes a wc_MlDsaKey with a device-side key identifier. Equivalent to wc_MlDsaKey_Init() but also stashes a binary id that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.

Parameters:

- **key** Pointer to the wc_MlDsaKey to initialize.
- **id** Pointer to the device-side key identifier bytes. May be NULL when len is 0.
- **len** Number of bytes in id; must be in [0, MLDSA_MAX_ID_LEN].
- **heap** Heap hint for dynamic memory allocation. May be NULL.
- **devId** Device identifier for the crypto callback. Should be a registered cb devId, not INVALID_DEVID.

See:

- wc_MlDsaKey_Init

- wc_MlDsaKey_InitLabel
- wc_MlDsaKey_Free

Return:

- 0 on success.
- BAD_FUNC_ARG if key is NULL, or if id is NULL and len is positive.
- BUFFER_E if len is negative or exceeds MLDSA_MAX_ID_LEN.

The id is copied into the key object; the caller may free its buffer immediately after this call returns.

The key is given the highest parameter set compiled in (WC_ML_DSA_87 unless it was disabled at build time), as the parameter set of the key held on the device is not known here. Call wc_MlDsaKey_SetParams() to select a different one.

C.63.2.5 function wc_MlDsaKey_InitLabel

```
int wc_MlDsaKey_InitLabel(
    wc_MlDsaKey * key,
    const char * label,
    void * heap,
    int devId
)
```

Initializes a wc_MlDsaKey with a device-side key label. Equivalent to wc_MlDsaKey_Init() but also stashes a label string that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.

Parameters:

- **key** Pointer to the wc_MlDsaKey to initialize.
- **label** NUL-terminated device-side key label string.
- **heap** Heap hint for dynamic memory allocation. May be NULL.
- **devId** Device identifier for the crypto callback.

See:

- wc_MlDsaKey_Init
- wc_MlDsaKey_InitId
- wc_MlDsaKey_Free

Return:

- 0 on success.
- BAD_FUNC_ARG if key or label is NULL.
- BUFFER_E if label is empty or longer than MLDSA_MAX_LABEL_LEN.

The label length is taken via XSTRLEN, so embedded NUL bytes terminate the label.

The parameter set is chosen as in wc_MlDsaKey_InitId().

C.63.2.6 function wc_MlDsaKey_SetParams

```
int wc_MlDsaKey_SetParams(
    wc_MlDsaKey * key,
    byte level
)
```

Selects the ML-DSA parameter set for this key. Must be called after wc_MlDsaKey_Init() and before key generation, signing or verifying.

Parameters:

- **key** Pointer to an initialized wc_MlDsaKey.
- **level** Parameter set: WC_ML_DSA_44, WC_ML_DSA_65 or WC_ML_DSA_87.

See:

- wc_MlDsaKey_GetParams
- wc_MlDsaKey_Init

Return:

- 0 on success.
- BAD_FUNC_ARG if key is NULL or level is not a recognized parameter set.
- NOT_COMPILED_IN if level names a parameter set that was disabled at build time.

C.63.2.7 function wc_MlDsaKey_GetParams

```
int wc_MlDsaKey_GetParams(
    wc_MlDsaKey * key,
    byte * level
)
```

Retrieves the ML-DSA parameter set currently selected on this key.

Parameters:

- **key** Pointer to an initialized wc_MlDsaKey.
- **level** Receives WC_ML_DSA_44, WC_ML_DSA_65 or WC_ML_DSA_87.

See: wc_MlDsaKey_SetParams

Return:

- 0 on success.
- BAD_FUNC_ARG if key or level is NULL.

C.63.2.8 function wc_MlDsaKey_Free

```
void wc_MlDsaKey_Free(
    wc_MlDsaKey * key
)
```

Releases resources held by a wc_MlDsaKey. After this call the object must be re-initialized with wc_MlDsaKey_Init() before it can be used again. Safe to call with a NULL pointer.

Parameters:

- **key** Pointer to the wc_MlDsaKey to free.

See: wc_MlDsaKey_Init

C.63.2.9 function wc_MlDsaKey_MakeKey

```
int wc_MlDsaKey_MakeKey(
    wc_MlDsaKey * key,
    WC_RNG * rng
)
```

Generates a new ML-DSA key pair using the provided RNG. The parameter set must already be set with wc_MlDsaKey_SetParams(). Both the public and private key components are populated on success.

Parameters:

- **key** Pointer to a wc_MlDsaKey with parameters set.

- **rng** Pointer to an initialized WC_RNG.

See:

- wc_MlDsaKey_MakeKeyFromSeed
- wc_MlDsaKey_SetParams

Return:

- 0 on success.
- BAD_FUNC_ARG if key or rng is NULL.
- MEMORY_E on allocation failure.

Example

```
wc_MlDsaKey key;
WC_RNG rng;

wc_MlDsaKey_Init(&key, NULL, INVALID_DEVID);
wc_MlDsaKey_SetParams(&key, WC_ML_DSA_65);
wc_InitRng(&rng);

if (wc_MlDsaKey_MakeKey(&key, &rng) != 0) {
    // error generating key pair
}
```

C.63.2.10 function wc_MlDsaKey_MakeKeyFromSeed

```
int wc_MlDsaKey_MakeKeyFromSeed(
    wc_MlDsaKey * key,
    const byte * seed
)
```

Deterministically generates an ML-DSA key pair from a 32-byte seed. Useful for known-answer tests and for applications that derive the seed from another secret. The seed buffer must contain exactly 32 bytes.

Parameters:

- **key** Pointer to a wc_MlDsaKey with parameters set.
- **seed** Pointer to a 32-byte seed buffer.

See: wc_MlDsaKey_MakeKey

Return:

- 0 on success.
- BAD_FUNC_ARG if key or seed is NULL.

C.63.2.11 function wc_MlDsaKey_MakePublicKey

```
int wc_MlDsaKey_MakePublicKey(
    wc_MlDsaKey * key
)
```

Derives public key for a wc_MlDsaKey with private key set. If the public key is already set, it is only checked against the private key's tr (H(pk)); use wc_MlDsaKey_CheckKey() for a full key pair check. The private key is still required: a public-only key returns BAD_FUNC_ARG. Available when WC_MLDSA_HAVE_MAKE_PUBLIC_KEY is defined: that is, when WOLFSSL_MLDSA_ASSIGN_KEY, WOLFSSL_MLDSA_NO_MAKE_KEY and WOLFSSL_MLDSA_VERIFY_ONLY are not defined.

Parameters:

- **key** Pointer to wc_MlDsaKey.

See:

- wc_MlDsaKey_ImportPrivRaw
- wc_MlDsaKey_MakeKey
- wc_MlDsaKey_CheckKey

Return:

- 0 on success or already set.
- BAD_FUNC_ARG if invalid args or the private key is not set.
- MEMORY_E on allocation failure.
- PUBLIC_KEY_E if the public key does not match the private key's tr or, when derived, its t0.
- NO_VALID_DEVID with WOLF_CRYPTO_CB_ONLY_MLDSA.
- Other negative on error.

Derivation is always done in software from the local private key bytes, including for a key initialized with a devId: a key whose private key was imported with wc_MlDsaKey_ImportPrivRaw() (or generated in software) derives normally. A key that only references a private key held on a device (wc_MlDsaKey_InitId() / wc_MlDsaKey_InitLabel()) has no local private bytes, so it returns BAD_FUNC_ARG. On PUBLIC_KEY_E an already set public key is left in place; set the matching public key, or re-initialize the key, before deriving.

C.63.2.12 function wc_MlDsaKey_SignCtx

```
int wc_MlDsaKey_SignCtx(
    wc_MlDsaKey * key,
    const byte * ctx,
    byte ctxLen,
    byte * sig,
    word32 * sigLen,
    const byte * msg,
    word32 msgLen,
    WC_RNG * rng
)
```

Signs a message with ML-DSA using the FIPS 204 randomized-with-context signing API. Pass ctx=NULL and ctxLen=0 for an empty context.

Parameters:

- **key** Pointer to a wc_MlDsaKey with the private key.
- **ctx** Optional context string (may be NULL when ctxLen=0).
- **ctxLen** Length of ctx in bytes; must be 0 when ctx is NULL, and no greater than 255.
- **sig** Buffer that receives the signature.
- **sigLen** In: size of sig. Out: bytes written.
- **msg** Message to sign.
- **msgLen** Length of msg in bytes.
- **rng** Pointer to an initialized WC_RNG.

See:

- wc_MlDsaKey_VerifyCtx
- wc_MlDsaKey_SignCtxWithSeed
- wc_MlDsaKey_SignCtxHash

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL, ctxLen is invalid or the key has no private key set.
- BUFFER_E if the sig buffer is too small.

On entry *sigLen is the size of the sig buffer; on success it is updated to the number of bytes written. Use wc_MlDsaKey_SigSize() or wc_MlDsaKey_GetSigLen() to determine the required buffer size.

C.63.2.13 function wc_MlDsaKey_SignCtxHash

```
int wc_MlDsaKey_SignCtxHash(
    wc_MlDsaKey * key,
    const byte * ctx,
    byte ctxLen,
    byte * sig,
    word32 * sigLen,
    const byte * hash,
    word32 hashLen,
    int hashAlg,
    WC_RNG * rng
)
```

HashML-DSA signing variant: signs a pre-hashed message. The caller supplies the hash bytes and identifies the hash algorithm. This is the “pre-hash” mode of FIPS 204.

Parameters:

- **key** Pointer to a wc_MlDsaKey with the private key.
- **ctx** Optional context string (NULL when ctxLen=0).
- **ctxLen** Length of ctx in bytes; no greater than 255.
- **sig** Buffer that receives the signature.
- **sigLen** In: size of sig. Out: bytes written.
- **hash** The message digest to sign.
- **hashLen** Length of hash in bytes.
- **hashAlg** Hash algorithm identifier (e.g. WC_HASH_TYPE_SHA256).
- **rng** Pointer to an initialized WC_RNG.

See:

- wc_MlDsaKey_SignCtx
- wc_MlDsaKey_VerifyCtxHash

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL, ctxLen is invalid, hashAlg is not supported or is weaker than the parameter set (FIPS 204 sec 5.4), or the key has no private key set.
- BUFFER_E if the sig buffer is too small.

Note: WOLFSSL_MLDSA_ALLOW_WEAK_PREHASH accepts a supported hash weaker than the parameter set instead of returning BAD_FUNC_ARG. An unsupported hash is still refused. Not FIPS 204 section 5.4 compliant.

C.63.2.14 function wc_MlDsaKey_Sign

```
int wc_MlDsaKey_Sign(
    wc_MlDsaKey * key,
    byte * sig,
    word32 * sigLen,
```

```
    const byte * msg,
    word32 msgLen,
    WC_RNG * rng
)
```

Legacy ML-DSA signing API without a context parameter. Only available when wolfSSL is built with WOLFSSL_MLDSA_NO_CTX. New code should call wc_MlDsaKey_SignCtx() with ctx=NULL and ctxLen=0 for a FIPS 204 compliant empty-context signature.

Parameters:

- **key** Pointer to a wc_MlDsaKey with the private key.
- **sig** Buffer that receives the signature.
- **sigLen** In: size of sig. Out: bytes written.
- **msg** Message to sign.
- **msgLen** Length of msg in bytes.
- **rng** Pointer to an initialized WC_RNG.

See:

- wc_MlDsaKey_SignCtx
- wc_MlDsaKey_Verify

Return: See wc_MlDsaKey_SignCtx().

C.63.2.15 function wc_MlDsaKey_SignCtxWithSeed

```
int wc_MlDsaKey_SignCtxWithSeed(
    wc_MlDsaKey * key,
    const byte * ctx,
    byte ctxLen,
    byte * sig,
    word32 * sigLen,
    const byte * msg,
    word32 msgLen,
    const byte * seed
)
```

Deterministic signing variant of wc_MlDsaKey_SignCtx(). The 32-byte seed replaces the randomness an RNG would supply, so the same key/ctx/msg/seed always produces the same signature.

Parameters:

- **key** Pointer to a wc_MlDsaKey with the private key.
- **ctx** Optional context string (NULL when ctxLen=0).
- **ctxLen** Length of ctx; no greater than 255.
- **sig** Buffer that receives the signature.
- **sigLen** In: size of sig. Out: bytes written.
- **msg** Message to sign.
- **msgLen** Length of msg in bytes.
- **seed** 32-byte seed bytes.

See:

- wc_MlDsaKey_SignCtx
- wc_MlDsaKey_SignCtxHashWithSeed

Return: See wc_MlDsaKey_SignCtx().

C.63.2.16 function wc_MlDsaKey_SignCtxHashWithSeed

```
int wc_MlDsaKey_SignCtxHashWithSeed(
    wc_MlDsaKey * key,
    const byte * ctx,
    byte ctxLen,
    byte * sig,
    word32 * sigLen,
    const byte * hash,
    word32 hashLen,
    int hashAlg,
    const byte * seed
)
```

Deterministic HashML-DSA signing: like wc_MlDsaKey_SignCtxHash() but uses the supplied 32-byte seed in place of an RNG.

Parameters:

- **key** Pointer to a wc_MlDsaKey with the private key.
- **ctx** Optional context string (NULL when ctxLen=0).
- **ctxLen** Length of ctx; no greater than 255.
- **sig** Buffer that receives the signature.
- **sigLen** In: size of sig. Out: bytes written.
- **hash** The message digest to sign.
- **hashLen** Length of hash in bytes.
- **hashAlg** Hash algorithm identifier.
- **seed** 32-byte seed bytes.

See: wc_MlDsaKey_SignCtxHash

Return: See wc_MlDsaKey_SignCtxHash().

C.63.2.17 function wc_MlDsaKey_SignMuWithSeed

```
int wc_MlDsaKey_SignMuWithSeed(
    wc_MlDsaKey * key,
    byte * sig,
    word32 * sigLen,
    const byte * mu,
    word32 muLen,
    const byte * seed
)
```

Signs a pre-computed mu value (the externally derived SHAKE256 hash of (tr || ctx || msg) per FIPS 204) using a deterministic 32-byte seed. Used by protocols that need to split the message-hashing step from the signing step.

Parameters:

- **key** Pointer to a wc_MlDsaKey with the private key.
- **sig** Buffer that receives the signature.
- **sigLen** In: size of sig. Out: bytes written.
- **mu** The 64-byte mu value (SHAKE256 output).
- **muLen** Length of mu; must be 64.
- **seed** 32-byte seed bytes.

See: wc_MlDsaKey_VerifyMu

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL, muLen is not 64 or the key has no private key set.
- BUFFER_E if the sig buffer is too small.

C.63.2.18 function wc_MlDsaKey_SignWithSeed

```
int wc_MlDsaKey_SignWithSeed(
    wc_MlDsaKey * key,
    byte * sig,
    word32 * sigLen,
    const byte * msg,
    word32 msgLen,
    const byte * seed
)
```

Legacy seed-based signing API without a context parameter. Only available when wolfSSL is built with WOLFSSL_MLDSA_NO_CTX. New code should use wc_MlDsaKey_SignCtxWithSeed().

Parameters:

- **key** Pointer to a wc_MlDsaKey with the private key.
- **sig** Buffer that receives the signature.
- **sigLen** In: size of sig. Out: bytes written.
- **msg** Message to sign.
- **msgLen** Length of msg in bytes.
- **seed** 32-byte seed bytes.

See: wc_MlDsaKey_SignCtxWithSeed

Return: See wc_MlDsaKey_SignCtxWithSeed().

C.63.2.19 function wc_MlDsaKey_VerifyCtx

```
int wc_MlDsaKey_VerifyCtx(
    wc_MlDsaKey * key,
    const byte * sig,
    word32 sigLen,
    const byte * ctx,
    byte ctxLen,
    const byte * msg,
    word32 msgLen,
    int * res
)
```

Verifies an ML-DSA signature produced by wc_MlDsaKey_SignCtx() or one of its variants. On entry res is set to 0; on success it is set to 1 when the signature is valid and left at 0 otherwise. The function's return value reports whether verification could be carried out at all; a bad signature is NOT a function-level error.

Parameters:

- **key** Pointer to a wc_MlDsaKey with the public key.
- **sig** Signature bytes to verify.
- **sigLen** Length of sig in bytes.
- **ctx** Optional context string (NULL when ctxLen=0).
- **ctxLen** Length of ctx; no greater than 255.
- **msg** Message that was signed.

- **msgLen** Length of msg in bytes.
- **res** Set to 1 on a valid signature, 0 otherwise.

See:

- wc_MlDsaKey_SignCtx
- wc_MlDsaKey_VerifyCtxHash
- wc_MlDsaKey_VerifyMu

Return:

- 0 if verification completed (check res for the result).
- BAD_FUNC_ARG if any required pointer is NULL or ctxLen is invalid.

C.63.2.20 function wc_MlDsaKey_VerifyCtxHash

```
int wc_MlDsaKey_VerifyCtxHash(
    wc_MlDsaKey * key,
    const byte * sig,
    word32 sigLen,
    const byte * ctx,
    byte ctxLen,
    const byte * hash,
    word32 hashLen,
    int hashAlg,
    int * res
)
```

Verifies a HashML-DSA signature where the message digest was supplied directly. See wc_MlDsaKey_VerifyCtx() for the semantics of res.

Parameters:

- **key** Pointer to a wc_MlDsaKey with the public key.
- **sig** Signature bytes to verify.
- **sigLen** Length of sig in bytes.
- **ctx** Optional context string (NULL when ctxLen=0).
- **ctxLen** Length of ctx; no greater than 255.
- **hash** The message digest that was signed.
- **hashLen** Length of hash in bytes.
- **hashAlg** Hash algorithm identifier.
- **res** Set to 1 on a valid signature, 0 otherwise.

See:

- wc_MlDsaKey_SignCtxHash
- wc_MlDsaKey_VerifyCtx

Return:

- 0 if verification completed (check res for the result).
- BAD_FUNC_ARG if any required pointer is NULL, ctxLen is invalid, or hashAlg is unsupported or weaker than the parameter set (FIPS 204 sec 5.4).

Note: WOLFSSL_MLDSA_ALLOW_WEAK_PREHASH verifies such a signature rather than refusing it. An unsupported hash is still refused.

C.63.2.21 function wc_MlDsaKey_VerifyMu

```
int wc_MlDsaKey_VerifyMu(
    wc_MlDsaKey * key,
    const byte * sig,
    word32 sigLen,
    const byte * mu,
    word32 muLen,
    int * res
)
```

Verifies a signature over a pre-computed 64-byte mu value (see wc_MlDsaKey_SignMuWithSeed()). See wc_MlDsaKey_VerifyCtx() for the semantics of res.

Parameters:

- **key** Pointer to a wc_MlDsaKey with the public key.
- **sig** Signature bytes to verify.
- **sigLen** Length of sig in bytes.
- **mu** The 64-byte mu value.
- **muLen** Length of mu; must be 64.
- **res** Set to 1 on a valid signature, 0 otherwise.

See: wc_MlDsaKey_SignMuWithSeed

Return:

- 0 if verification completed (check res for the result).
- BAD_FUNC_ARG if any required pointer is NULL or muLen is not 64.

C.63.2.22 function wc_MlDsaKey_SetPrecompA

```
int wc_MlDsaKey_SetPrecompA(
    wc_MlDsaKey * key,
    const sword32 * a,
    word32 aLen,
    const byte * rho,
    word32 rhoLen
)
```

Attaches a matrix A that was expanded off target, so verification can skip the SHAKE128 rejection sampling that otherwise dominates it. A is a function of the public seed rho alone, so wherever the verification key is fixed at build time - secure boot being the usual case - A can be computed once on a host and stored in flash. Available only when WOLFSSL_MLDSA_VERIFY_PRECOMP_A is defined, and honoured by both the default and the small-memory verifiers.

Parameters:

- **key** Pointer to a wc_MlDsaKey holding the public key.
- **a** Expanded matrix A, k * l * MLDSA_N elements, caller owned.
- **aLen** Number of sword32 elements in a.
- **rho** Public seed the matrix was expanded from.
- **rhoLen** Length of rho; must be MLDSA_PUB_SEED_SZ.

See:

- wc_MlDsaKey_VerifyMu
- wc_MlDsaKey_ImportPubRaw

Return:

- 0 on success.

- BAD_FUNC_ARG if a pointer is NULL, rhoLen is not MLDSA_PUB_SEED_SZ, aLen is not k * l * MLDSA_N, or no public key has been imported yet.
- PUBLIC_KEY_E if rho does not match the imported public key.

Note: A is derived from the public key and must be integrity-protected exactly as the public key is. An attacker able to substitute it can influence verification. In a secure-boot design the key already lives in protected flash, so store the two together.

The matrix is borrowed, never copied and never freed: it must stay valid and unchanged for as long as the key uses it. It holds k * l * MLDSA_N sword32 elements (not bytes) in row-major (r, s) order and in the NTT domain, exactly as ExpandA produces them.

Import the public key first: the matrix is bound to it by comparing rho. The binding is re-checked on every verify, so replacing the key or the parameters afterwards simply discards the matrix and falls back to expanding A rather than using a stale one.

Example

```
wc_MlDsaKey key;
int res = 0;

wc_MlDsaKey_Init(&key, NULL, INVALID_DEVID);
wc_MlDsaKey_SetParams(&key, WC_ML_DSA_87);
wc_MlDsaKey_ImportPubRaw(&key, pub, pubLen);
if (wc_MlDsaKey_SetPrecompA(&key, matrixA, matrixALen, pub,
        MLDSA_PUB_SEED_SZ) != 0) {
    // handle error
}
wc_MlDsaKey_VerifyMu(&key, sig, sigLen, mu, muLen, &res);
```

C.63.2.23 function wc_MlDsaKey_Verify

```
int wc_MlDsaKey_Verify(
    wc_MlDsaKey * key,
    const byte * sig,
    word32 sigLen,
    const byte * msg,
    word32 msgLen,
    int * res
)
```

Legacy ML-DSA verify API without a context parameter. Only available when wolfSSL is built with WOLFSSL_MLDSA_NO_CTX. New code should use wc_MlDsaKey_VerifyCtx() with ctx=NULL and ctxLen=0.

Parameters:

- **key** Pointer to a wc_MlDsaKey with the public key.
- **sig** Signature bytes to verify.
- **sigLen** Length of sig in bytes.
- **msg** Message that was signed.
- **msgLen** Length of msg in bytes.
- **res** Set to 1 on a valid signature, 0 otherwise.

See:

- wc_MlDsaKey_VerifyCtx
- wc_MlDsaKey_Sign

Return: See wc_MlDsaKey_VerifyCtx().

C.63.2.24 function wc_MlDsaKey_Size

```
int wc_MlDsaKey_Size(
    wc_MlDsaKey * key
)
```

Returns the size in bytes of the encoded private key for the parameter set selected on this key. Equivalent to wc_MlDsaKey_PrivSize() and provided for API compatibility.

Parameters:

- **key** Pointer to a wc_MlDsaKey with parameters set.

See:

- wc_MlDsaKey_PrivSize
- wc_MlDsaKey_PubSize
- wc_MlDsaKey_SigSize

Return:

- Encoded private key size in bytes on success (positive value).
- BAD_FUNC_ARG if key is NULL or no parameter set is selected.

C.63.2.25 function wc_MlDsaKey_PrivSize

```
int wc_MlDsaKey_PrivSize(
    wc_MlDsaKey * key
)
```

Returns the size in bytes of the encoded private key for the parameter set selected on this key.

Parameters:

- **key** Pointer to a wc_MlDsaKey with parameters set.

See:

- wc_MlDsaKey_PubSize
- wc_MlDsaKey_GetPrivLen

Return:

- Encoded private key size on success (positive value).
- BAD_FUNC_ARG if key is NULL or no parameter set is selected.

C.63.2.26 function wc_MlDsaKey_PubSize

```
int wc_MlDsaKey_PubSize(
    wc_MlDsaKey * key
)
```

Returns the size in bytes of the encoded public key for the parameter set selected on this key.

Parameters:

- **key** Pointer to a wc_MlDsaKey with parameters set.

See:

- wc_MlDsaKey_PrivSize
- wc_MlDsaKey_GetPubLen

Return:

- Encoded public key size on success (positive value).
- BAD_FUNC_ARG if key is NULL or no parameter set is selected.

C.63.2.27 function wc_MlDsaKey_SigSize

```
int wc_MlDsaKey_SigSize(
    wc_MlDsaKey * key
)
```

Returns the size in bytes of the signature produced by this key's parameter set.

Parameters:

- **key** Pointer to a wc_MlDsaKey with parameters set.

See:

- wc_MlDsaKey_GetSigLen
- wc_MlDsaKey_SignCtx

Return:

- Signature size on success (positive value).
- BAD_FUNC_ARG if key is NULL or no parameter set is selected.

C.63.2.28 function wc_MlDsaKey_GetPrivLen

```
int wc_MlDsaKey_GetPrivLen(
    wc_MlDsaKey * key,
    int * len
)
```

Writes the encoded private key size into *len. Equivalent information to wc_MlDsaKey_PrivSize() but uses an out-parameter style.

Parameters:

- **key** Pointer to a wc_MlDsaKey with parameters set.
- **len** Receives the private key size in bytes.

See: wc_MlDsaKey_PrivSize

Return:

- 0 on success.
- BAD_FUNC_ARG if key or len is NULL, or no parameter set is selected.

C.63.2.29 function wc_MlDsaKey_GetPubLen

```
int wc_MlDsaKey_GetPubLen(
    wc_MlDsaKey * key,
    int * len
)
```

Writes the encoded public key size into *len. Equivalent information to wc_MlDsaKey_PubSize() but uses an out-parameter style.

Parameters:

- **key** Pointer to a wc_MlDsaKey with parameters set.
- **len** Receives the public key size in bytes.

See: wc_MlDsaKey_PubSize

Return:

- 0 on success.
- BAD_FUNC_ARG if key or len is NULL, or no parameter set is selected.

C.63.2.30 function wc_MlDsaKey_GetSigLen

```
int wc_MlDsaKey_GetSigLen(
    wc_MlDsaKey * key,
    int * len
)
```

Writes the signature size into *len. Equivalent information to wc_MlDsaKey_SigSize() but uses an out-parameter style.

Parameters:

- **key** Pointer to a wc_MlDsaKey with parameters set.
- **len** Receives the signature size in bytes.

See: wc_MlDsaKey_SigSize

Return:

- 0 on success.
- BAD_FUNC_ARG if key or len is NULL, or no parameter set is selected.

C.63.2.31 function wc_MlDsaKey_CheckKey

```
int wc_MlDsaKey_CheckKey(
    wc_MlDsaKey * key
)
```

Self-checks an ML-DSA key by recomputing the public key from the private and comparing against the stored public. Only available when wolfSSL is built with WOLFSSL_MLDSA_CHECK_KEY.

Parameters:

- **key** Pointer to a wc_MlDsaKey with both public and private parts populated.

Return:

- 0 if the key is consistent.
- BAD_FUNC_ARG if key is NULL.
- PUBLIC_KEY_E if the recomputed public does not match.

C.63.2.32 function wc_MlDsaKey_ImportPubRaw

```
int wc_MlDsaKey_ImportPubRaw(
    wc_MlDsaKey * key,
    const byte * in,
    word32 inLen
)
```

Imports a raw ML-DSA public key. The parameter set must already be selected on the key. inLen must match the size returned by wc_MlDsaKey_PubSize() for the selected parameter set.

Parameters:

- **key** Pointer to a wc_MlDsaKey with parameters set.

- **in** Raw public key bytes.
- **inLen** Length of in in bytes.

See:

- wc_MlDsaKey_ExportPubRaw
- wc_MlDsaKey_ImportPrivRaw

Return:

- 0 on success.
- BAD_FUNC_ARG if key or in is NULL.
- BUFFER_E if inLen does not match the expected public key size.

C.63.2.33 function wc_MlDsaKey_ImportPrivRaw

```
int wc_MlDsaKey_ImportPrivRaw(
    wc_MlDsaKey * key,
    const byte * priv,
    word32 privSz
)
```

Imports a raw ML-DSA private key. The parameter set must already be selected. privSz must match the size returned by wc_MlDsaKey_PrivSize().

Parameters:

- **key** Pointer to a wc_MlDsaKey with parameters set.
- **priv** Raw private key bytes.
- **privSz** Length of priv in bytes.

See:

- wc_MlDsaKey_ExportPrivRaw
- wc_MlDsaKey_ImportKey

Return:

- 0 on success.
- BAD_FUNC_ARG if key or priv is NULL.
- BUFFER_E if privSz does not match the expected private key size.

C.63.2.34 function wc_MlDsaKey_ImportKey

```
int wc_MlDsaKey_ImportKey(
    wc_MlDsaKey * key,
    const byte * priv,
    word32 privSz,
    const byte * pub,
    word32 pubSz
)
```

Imports a raw ML-DSA key pair (private and public parts together). The parameter set must already be selected.

Parameters:

- **key** Pointer to a wc_MlDsaKey with parameters set.
- **priv** Raw private key bytes.
- **privSz** Length of priv.
- **pub** Raw public key bytes.

- **pubSz** Length of pub.

See: wc_MlDsaKey_ExportKey

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- BUFFER_E if privSz or pubSz does not match the expected sizes.

C.63.2.35 function wc_MlDsaKey_ExportPubRaw

```
int wc_MlDsaKey_ExportPubRaw(
    wc_MlDsaKey * key,
    byte * out,
    word32 * outLen
)
```

Exports the raw ML-DSA public key. On entry *outLen is the size of out; on success it is updated to the bytes written.

Parameters:

- **key** Pointer to a wc_MlDsaKey with a public key.
- **out** Buffer that receives the public key.
- **outLen** In: size of out. Out: bytes written.

See: wc_MlDsaKey_ImportPubRaw

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- BUFFER_E if *outLen is smaller than the public key size.

C.63.2.36 function wc_MlDsaKey_ExportPrivRaw

```
int wc_MlDsaKey_ExportPrivRaw(
    wc_MlDsaKey * key,
    byte * out,
    word32 * outLen
)
```

Exports the raw ML-DSA private key. On entry *outLen is the size of out; on success it is updated to the bytes written.

Parameters:

- **key** Pointer to a wc_MlDsaKey with a private key.
- **out** Buffer that receives the private key.
- **outLen** In: size of out. Out: bytes written.

See: wc_MlDsaKey_ImportPrivRaw

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- BUFFER_E if *outLen is smaller than the private key size.

C.63.2.37 function wc_MlDsaKey_ExportKey

```
int wc_MlDsaKey_ExportKey(
    wc_MlDsaKey * key,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz
)
```

Exports both raw public and private ML-DSA key components in one call.

Parameters:

- **key** Pointer to a wc_MlDsaKey with both key parts.
- **priv** Buffer that receives the private key.
- **privSz** In: size of priv. Out: bytes written.
- **pub** Buffer that receives the public key.
- **pubSz** In: size of pub. Out: bytes written.

See: wc_MlDsaKey_ImportKey

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- BUFFER_E if either buffer is too small.

C.63.2.38 function wc_MlDsaKey_PrivateKeyDecode

```
int wc_MlDsaKey_PrivateKeyDecode(
    wc_MlDsaKey * key,
    const byte * input,
    word32 inSz,
    word32 * inOutIdx
)
```

Parses an ML-DSA private key from a DER/ASN.1 encoded buffer (PKCS#8 OneAsymmetricKey). The parameter set is inferred from the algorithm identifier in the encoding, so it does NOT need to be set beforehand. On success *inOutIdx is advanced past the consumed bytes.

Parameters:

- **key** Pointer to an initialized wc_MlDsaKey.
- **input** DER-encoded private key bytes.
- **inSz** Length of input in bytes.
- **inOutIdx** In: offset into input where decoding starts. Out: offset past the consumed bytes.

See:

- wc_MlDsaKey_PrivateKeyToDer
- wc_MlDsaKey_PublicKeyDecode

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- ASN_PARSE_E on malformed encoding.

Only available when WOLFSSL_MLDSA_NO_ASN1 is not defined.

C.63.2.39 function wc_MlDsaKey_PublicKeyDecode

```
int wc_MlDsaKey_PublicKeyDecode(
    wc_MlDsaKey * key,
    const byte * input,
    word32 inSz,
    word32 * inOutIdx
)
```

Parses an ML-DSA public key from a DER/ASN.1 encoded buffer (SubjectPublicKeyInfo). The parameter set is inferred from the algorithm identifier. On success *inOutIdx is advanced past the consumed bytes.

Parameters:

- **key** Pointer to an initialized wc_MlDsaKey.
- **input** DER-encoded SPKI bytes.
- **inSz** Length of input in bytes.
- **inOutIdx** In: offset into input where decoding starts. Out: offset past the consumed bytes.

See: wc_MlDsaKey_PublicKeyToDer

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- ASN_PARSE_E on malformed encoding.

C.63.2.40 function wc_MlDsaKey_PublicKeyToDer

```
int wc_MlDsaKey_PublicKeyToDer(
    wc_MlDsaKey * key,
    byte * output,
    word32 inLen,
    int withAlg
)
```

Encodes an ML-DSA public key to DER. When withAlg is non-zero the output is a full SubjectPublicKeyInfo (with AlgorithmIdentifier); when zero the output is the raw public key bytes.

Parameters:

- **key** Pointer to a wc_MlDsaKey with a public key.
- **output** Buffer that receives the DER encoding, or NULL to query size.
- **inLen** Size of output (ignored when output is NULL).
- **withAlg** Non-zero to emit SubjectPublicKeyInfo, zero for the raw public key only.

See:

- wc_MlDsaKey_PublicKeyDecode
- wc_MlDsaKey_KeyToDer

Return:

- Size of the encoded DER in bytes on success.
- BAD_FUNC_ARG if key is NULL or no parameter set is selected.
- BUFFER_E if output is non-NULL and inLen is smaller than the required size.

Pass NULL as output to query the required buffer size.

C.63.2.41 function wc_MlDsaKey_KeyToDer

```
int wc_MlDsaKey_KeyToDer(
    wc_MlDsaKey * key,
    byte * output,
    word32 inLen
)
```

Encodes an ML-DSA key pair (public + private) to DER as a PKCS#8 OneAsymmetricKey structure. Pass NULL as output to query the required buffer size.

Parameters:

- **key** Pointer to a wc_MlDsaKey with the private key.
- **output** Buffer that receives the DER encoding, or NULL to query size.
- **inLen** Size of output (ignored when output is NULL).

See:

- wc_MlDsaKey_PrivateKeyDecode
- wc_MlDsaKey_PrivateKeyToDer
- wc_MlDsaKey_PublicKeyToDer

Return:

- Size of the encoded DER in bytes on success.
- BAD_FUNC_ARG if key is NULL or no parameter set is selected.
- MISSING_KEY if the private key has not been set.
- BUFFER_E if output is non-NULL and inLen is too small.

C.63.2.42 function wc_MlDsaKey_PrivateKeyToDer

```
int wc_MlDsaKey_PrivateKeyToDer(
    wc_MlDsaKey * key,
    byte * output,
    word32 inLen
)
```

Encodes the ML-DSA private key to DER. Per FIPS 204 the private key encoding includes the public component, so this function is currently an alias of wc_MlDsaKey_KeyToDer() kept for API parity with other algorithms.

Parameters:

- **key** Pointer to a wc_MlDsaKey with the private key.
- **output** Buffer that receives the DER encoding, or NULL to query size.
- **inLen** Size of output (ignored when output is NULL).

See:

- wc_MlDsaKey_KeyToDer
- wc_MlDsaKey_PrivateKeyDecode

Return:

- Size of the encoded DER in bytes on success.
- Inherited error codes from wc_MlDsaKey_KeyToDer().

C.63.3 Source code

```
int wc_MlDsaKey_Init(wc_MlDsaKey* key, void* heap, int devId);
```

```
wc_MlDsaKey* wc_MlDsaKey_New(void* heap, int devId);

int wc_MlDsaKey_Delete(wc_MlDsaKey* key, wc_MlDsaKey** key_p);

int wc_MlDsaKey_InitId(wc_MlDsaKey* key, const unsigned char* id, int len,
    void* heap, int devId);

int wc_MlDsaKey_InitLabel(wc_MlDsaKey* key, const char* label, void* heap,
    int devId);

int wc_MlDsaKey_SetParams(wc_MlDsaKey* key, byte level);

int wc_MlDsaKey_GetParams(wc_MlDsaKey* key, byte* level);

void wc_MlDsaKey_Free(wc_MlDsaKey* key);

int wc_MlDsaKey_MakeKey(wc_MlDsaKey* key, WC_RNG* rng);

int wc_MlDsaKey_MakeKeyFromSeed(wc_MlDsaKey* key, const byte* seed);

int wc_MlDsaKey_MakePublicKey(wc_MlDsaKey* key);

int wc_MlDsaKey_SignCtx(wc_MlDsaKey* key, const byte* ctx, byte ctxLen,
    byte* sig, word32* sigLen, const byte* msg, word32 msgLen, WC_RNG* rng);

int wc_MlDsaKey_SignCtxHash(wc_MlDsaKey* key, const byte* ctx, byte ctxLen,
    byte* sig, word32* sigLen, const byte* hash, word32 hashLen,
    int hashAlg, WC_RNG* rng);

int wc_MlDsaKey_Sign(wc_MlDsaKey* key, byte* sig, word32* sigLen,
    const byte* msg, word32 msgLen, WC_RNG* rng);

int wc_MlDsaKey_SignCtxWithSeed(wc_MlDsaKey* key, const byte* ctx, byte ctxLen,
    byte* sig, word32* sigLen, const byte* msg, word32 msgLen,
    const byte* seed);

int wc_MlDsaKey_SignCtxHashWithSeed(wc_MlDsaKey* key, const byte* ctx,
    byte ctxLen, byte* sig, word32* sigLen, const byte* hash,
    word32 hashLen, int hashAlg, const byte* seed);

int wc_MlDsaKey_SignMuWithSeed(wc_MlDsaKey* key, byte* sig, word32* sigLen,
    const byte* mu, word32 muLen, const byte* seed);

int wc_MlDsaKey_SignWithSeed(wc_MlDsaKey* key, byte* sig, word32* sigLen,
    const byte* msg, word32 msgLen, const byte* seed);

int wc_MlDsaKey_VerifyCtx(wc_MlDsaKey* key, const byte* sig, word32 sigLen,
    const byte* ctx, byte ctxLen, const byte* msg, word32 msgLen, int* res);

int wc_MlDsaKey_VerifyCtxHash(wc_MlDsaKey* key, const byte* sig, word32 sigLen,
    const byte* ctx, byte ctxLen, const byte* hash, word32 hashLen,
    int hashAlg, int* res);
```

```
int wc_MlDsaKey_VerifyMu(wc_MlDsaKey* key, const byte* sig, word32 sigLen,
    const byte* mu, word32 muLen, int* res);

int wc_MlDsaKey_SetPrecompA(wc_MlDsaKey* key, const sword32* a, word32 aLen,
    const byte* rho, word32 rhoLen);

int wc_MlDsaKey_Verify(wc_MlDsaKey* key, const byte* sig, word32 sigLen,
    const byte* msg, word32 msgLen, int* res);

int wc_MlDsaKey_Size(wc_MlDsaKey* key);

int wc_MlDsaKey_PrivSize(wc_MlDsaKey* key);

int wc_MlDsaKey_PubSize(wc_MlDsaKey* key);

int wc_MlDsaKey_SigSize(wc_MlDsaKey* key);

int wc_MlDsaKey_GetPrivLen(wc_MlDsaKey* key, int* len);

int wc_MlDsaKey_GetPubLen(wc_MlDsaKey* key, int* len);

int wc_MlDsaKey_GetSigLen(wc_MlDsaKey* key, int* len);

int wc_MlDsaKey_CheckKey(wc_MlDsaKey* key);

int wc_MlDsaKey_ImportPubRaw(wc_MlDsaKey* key, const byte* in, word32 inLen);

int wc_MlDsaKey_ImportPrivRaw(wc_MlDsaKey* key, const byte* priv,
    word32 privSz);

int wc_MlDsaKey_ImportKey(wc_MlDsaKey* key, const byte* priv, word32 privSz,
    const byte* pub, word32 pubSz);

int wc_MlDsaKey_ExportPubRaw(wc_MlDsaKey* key, byte* out, word32* outLen);

int wc_MlDsaKey_ExportPrivRaw(wc_MlDsaKey* key, byte* out, word32* outLen);

int wc_MlDsaKey_ExportKey(wc_MlDsaKey* key, byte* priv, word32 *privSz,
    byte* pub, word32 *pubSz);

int wc_MlDsaKey_PrivateKeyDecode(wc_MlDsaKey* key, const byte* input,
    word32 inSz, word32* inOutIdx);

int wc_MlDsaKey_PublicKeyDecode(wc_MlDsaKey* key, const byte* input,
    word32 inSz, word32* inOutIdx);

int wc_MlDsaKey_PublicKeyToDer(wc_MlDsaKey* key, byte* output,
    word32 inLen, int withAlg);

int wc_MlDsaKey_KeyToDer(wc_MlDsaKey* key, byte* output, word32 inLen);

int wc_MlDsaKey_PrivateKeyToDer(wc_MlDsaKey* key, byte* output,
    word32 inLen);
```

C.64 wc_mlkem.h

C.64.1 Functions

	Name
MlKemKey *	**wc_MlKemKey_New**(int type, void * heap, int devId)Allocates and initializes a new MlKemKey on the heap. The returned pointer must be released with wc_MlKemKey_Delete().
int	**wc_MlKemKey_Delete**(MlKemKey * key, MlKemKey ** key_p)Frees and zeros a heap_allocated MlKemKey previously returned by wc_MlKemKey_New(). On success the caller's pointer variable is set to NULL via key_p when key_p is not NULL.
int	**wc_MlKemKey_Init**(MlKemKey * key, int type, void * heap, int devId)Initializes an MlKemKey object in place. The type parameter selects the ML-KEM variant and must be one of WC_ML_KEM_512, WC_ML_KEM_768 or WC_ML_KEM_1024.
int	**wc_MlKemKey_Free**(MlKemKey * key)Releases resources held by an MlKemKey. After this call the object must be re_initialized with wc_MlKemKey_Init() before it can be used again. Safe to call with a NULL pointer.
int	**wc_MlKemKey_Init_Id**(MlKemKey * key, int type, const unsigned char * id, int len, void * heap, int devId)Initializes an MlKemKey with a device_side key identifier. Equivalent to wc_MlKemKey_Init() but also stashes a binary id that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.
int	**wc_MlKemKey_Init_Label**(MlKemKey * key, int type, const char * label, void * heap, int devId)Initializes an MlKemKey with a device_side key label. Equivalent to wc_MlKemKey_Init() but also stashes a label string that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.
int	**wc_MlKemKey_MakeKey**(MlKemKey * key, WC_RNG * rng)Generates a new ML-KEM key pair using the provided RNG. Both the public and private components are populated on success.

	Name
int	**wc_MlKemKey_MakeKeyWithRandom**(MlKemKey * key, const unsigned char * rand, int len)Deterministic key generation: produces an ML-KEM key pair from the supplied 64 bytes of randomness instead of an RNG. Useful for known-answer tests and for applications that derive key randomness from another secret.
int	**wc_MlKemKey_CipherTextSize**(MlKemKey * key, word32 * len)Returns the ciphertext size in bytes for the variant selected on this key.
int	**wc_MlKemKey_SharedSecretSize**(MlKemKey * key, word32 * len)Returns the shared_secret size in bytes for ML_KEM. The value is the same (32 bytes) across all parameter sets but is queried programmatically for symmetry with wc_MlKemKey_CipherTextSize().
int	**wc_MlKemKey_Encapsulate**(MlKemKey * key, unsigned char * ct, unsigned char * ss, WC_RNG * rng)Encapsulates a fresh shared secret against the public key held in key. Produces a ciphertext that the holder of the corresponding private key can pass to wc_MlKemKey_Decapsulate() to recover the same shared secret.
int	**wc_MlKemKey_EncapsulateWithRandom**(MlKemKey * key, unsigned char * ct, unsigned char * ss, const unsigned char * rand, int len)Deterministic variant of wc_MlKemKey_Encapsulate(). Uses the supplied 32 bytes of randomness instead of consuming output from an RNG.
int	**wc_MlKemKey_Decapsulate**(MlKemKey * key, unsigned char * ss, const unsigned char * ct, word32 len)Decapsulates a ciphertext using the private key held in key and recovers the shared secret produced by wc_MlKemKey_Encapsulate(). ML_KEM decapsulation is constant time and includes an implicit_rejection check on malformed ciphertexts (an attacker cannot learn the validity of ct from the runtime).
int	**wc_MlKemKey_DecodePrivateKey**(MlKemKey * key, const unsigned char * in, word32 len)Decodes a raw ML_KEM private key into key. The variant must already be selected on the key (typically via wc_MlKemKey_Init() or wc_MlKemKey_New()) and len must match the private key size for that variant.

	Name
int	**wc_MlKemKey_DecodePublicKey**(MlKemKey * key, const unsigned char * in, word32 len)Decodes a raw ML-KEM public key into key. The variant must already be selected on the key and len must match the public key size for that variant.
int	**wc_MlKemKey_PrivateKeySize**(MlKemKey * key, word32 * len)Returns the encoded private key size in bytes for the variant selected on this key.
int	**wc_MlKemKey_PublicKeySize**(MlKemKey * key, word32 * len)Returns the encoded public key size in bytes for the variant selected on this key.
int	**wc_MlKemKey_EncodePrivateKey**(MlKemKey * key, unsigned char * out, word32 len)Encodes the ML_KEM private key into out. The out buffer length must be exactly wc_MlKemKey_PrivateKeySize() bytes.
int	**wc_MlKemKey_EncodePublicKey**(MlKemKey * key, unsigned char * out, word32 len)Encodes the ML_KEM public key into out. The out buffer length must be exactly wc_MlKemKey_PublicKeySize() bytes.
int	**wc_MlKemKey_PublicKeyToDer**(MlKemKey * key, byte * output, word32 len, int withAlg)Encodes the ML-KEM public key as a DER SubjectPublicKeyInfo. Pass NULL for output to obtain the required length.
int	**wc_MlKemKey_PrivateKeyToDer**(MlKemKey * key, byte * output, word32 len)Encodes the ML-KEM private key as a DER PKCS#8 OneAsymmetricKey, carrying the expanded decapsulation key. Pass NULL for output to obtain the required length.
int	**wc_MlKemKey_PublicKeyDecode**(MlKemKey * key, const byte * input, word32 inSz, word32 * inOutIdx)Decodes a DER SubjectPublicKeyInfo into an ML-KEM public key. Takes wrapped DER, where wc_MlKemKey_DecodePublicKey takes the raw encoded key. A key initialized for a parameter set holds the DER to that same one. A key initialized with WC_ML_KEM_TYPE_UNSET takes the parameter set from the algorithm OID in the DER instead.

	Name
int	**wc_MlKemKey_PrivateKeyDecode**(MlKemKey * key, const byte * input, word32 inSz, word32 * inOutIdx)Decodes a DER PKCS#8 OneAsymmetricKey into an ML_KEM private key. Takes wrapped DER, where wc_MlKemKey_DecodePrivateKey takes the raw encoded key. All three RFC 9935 Section 6 CHOICE forms are accepted: the 64_byte seed under an implicit [0], the expanded decapsulation key as an OCTET STRING, and the SEQUENCE carrying both. A seed is expanded with ML_KEM.KeyGen_internal(d,z). For the “both” form the expanded key is regenerated from the seed and the two are compared, so a key whose halves disagree is rejected as malformed, as RFC 9935 Section 8 requires. A key initialized for a parameter set holds the DER to that same parameter set; a key initialized with WC_ML_KEM_TYPE_UNSET takes it from the algorithm OID in the DER instead.

C.64.2 Functions Documentation

C.64.2.1 function wc_MlKemKey_New

```
MlKemKey * wc_MlKemKey_New(
    int type,
    void * heap,
    int devId
)
```

Allocates and initializes a new MlKemKey on the heap. The returned pointer must be released with wc_MlKemKey_Delete().

Parameters:

- **type** ML-KEM variant: WC_ML_KEM_512, WC_ML_KEM_768 or WC_ML_KEM_1024.
- **heap** Heap hint for dynamic memory allocation. May be NULL.
- **devId** Device identifier for hardware crypto callbacks. Use INVALID_DEVID for software-only.

See:

- wc_MlKemKey_Delete
- wc_MlKemKey_Init

Return:

- Pointer to a freshly allocated MlKemKey on success.
- NULL on allocation failure or if type is invalid.

ML-KEM (FIPS 203) is a quantum-resistant key encapsulation mechanism. The type parameter selects the variant: WC_ML_KEM_512 (NIST security level 1), WC_ML_KEM_768 (level 3) or WC_ML_KEM_1024 (level 5).

Example

```
MlKemKey* key = wc_MlKemKey_New(WC_ML_KEM_768, NULL,
    INVALID_DEVID);
if (key == NULL) {
    // allocation failed
}
// ... use key ...
wc_MlKemKey_Delete(key, &key);
```

C.64.2.2 function wc_MlKemKey_Delete

```
int wc_MlKemKey_Delete(
    MlKemKey * key,
    MlKemKey ** key_p
)
```

Frees and zeros a heap-allocated MlKemKey previously returned by wc_MlKemKey_New(). On success the caller's pointer variable is set to NULL via key_p when key_p is not NULL.

Parameters:

- **key** The MlKemKey to free.
- **key_p** Optional address of the caller's pointer variable; when not NULL, it is set to NULL on success.

See: wc_MlKemKey_New

Return:

- 0 on success.
- BAD_FUNC_ARG if key is NULL.

C.64.2.3 function wc_MlKemKey_Init

```
int wc_MlKemKey_Init(
    MlKemKey * key,
    int type,
    void * heap,
    int devId
)
```

Initializes an MlKemKey object in place. The type parameter selects the ML-KEM variant and must be one of WC_ML_KEM_512, WC_ML_KEM_768 or WC_ML_KEM_1024.

Parameters:

- **key** Pointer to the MlKemKey to initialize.
- **type** ML-KEM variant: WC_ML_KEM_512, WC_ML_KEM_768 or WC_ML_KEM_1024. Pass WC_ML_KEM_TYPE_UNSET to set the object up without a parameter set, leaving wc_MlKemKey_PublicKeyDecode or wc_MlKemKey_PrivateKeyDecode to take it from the algorithm OID in the DER. Every other operation needs a parameter set and fails with BAD_FUNC_ARG until a decode has named one.
- **heap** Heap hint for dynamic memory allocation. May be NULL.
- **devId** Device identifier for hardware crypto callbacks.

See:

- wc_MlKemKey_Free
- wc_MlKemKey_MakeKey

Return:

- 0 on success.

- BAD_FUNC_ARG if key is NULL or type is invalid.
- NOT_COMPILED_IN if type names a variant that was disabled at build time.

Example

```
MlKemKey key;
int ret;

ret = wc_MlKemKey_Init(&key, WC_ML_KEM_768, NULL, INVALID_DEVID);
if (ret != 0) {
    // error initializing key
}
// ... use key ...
wc_MlKemKey_Free(&key);
```

C.64.2.4 function wc_MlKemKey_Free

```
int wc_MlKemKey_Free(
    MlKemKey * key
)
```

Releases resources held by an MlKemKey. After this call the object must be re-initialized with wc_MlKemKey_Init() before it can be used again. Safe to call with a NULL pointer.

Parameters:

- **key** Pointer to the MlKemKey to free.

See: wc_MlKemKey_Init

Return: 0 on success, including when key is NULL.

C.64.2.5 function wc_MlKemKey_Init_Id

```
int wc_MlKemKey_Init_Id(
    MlKemKey * key,
    int type,
    const unsigned char * id,
    int len,
    void * heap,
    int devId
)
```

Initializes an MlKemKey with a device-side key identifier. Equivalent to wc_MlKemKey_Init() but also stashes a binary id that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.

Parameters:

- **key** Pointer to the MlKemKey to initialize.
- **type** ML-KEM variant identifier.
- **id** Pointer to the device-side key identifier bytes. May be NULL when len is 0.
- **len** Number of bytes in id. Must be in the range [0, MLKEM_MAX_ID_LEN].
- **heap** Heap hint for dynamic memory allocation.
- **devId** Device identifier for the crypto callback.

See:

- wc_MlKemKey_Init
- wc_MlKemKey_Init_Label

- wc_MlKemKey_Free

Return:

- 0 on success.
- BAD_FUNC_ARG if key is NULL, if id is NULL while len is non-zero, or if type is invalid.
- BUFFER_E if len is negative or exceeds MLKEM_MAX_ID_LEN.

The id is copied into the key object; the caller may free its buffer immediately after this call returns.

C.64.2.6 function wc_MlKemKey_Init_Label

```
int wc_MlKemKey_Init_Label(
    MlKemKey * key,
    int type,
    const char * label,
    void * heap,
    int devId
)
```

Initializes an MlKemKey with a device-side key label. Equivalent to wc_MlKemKey_Init() but also stashes a label string that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.

Parameters:

- **key** Pointer to the MlKemKey to initialize.
- **type** ML-KEM variant identifier.
- **label** NUL-terminated device-side key label.
- **heap** Heap hint for dynamic memory allocation.
- **devId** Device identifier for the crypto callback.

See:

- wc_MlKemKey_Init
- wc_MlKemKey_Init_Id
- wc_MlKemKey_Free

Return:

- 0 on success.
- BAD_FUNC_ARG if key or label is NULL or type is invalid.

C.64.2.7 function wc_MlKemKey_MakeKey

```
int wc_MlKemKey_MakeKey(
    MlKemKey * key,
    WC_RNG * rng
)
```

Generates a new ML-KEM key pair using the provided RNG. Both the public and private components are populated on success.

Parameters:

- **key** Pointer to an initialized MlKemKey.
- **rng** Pointer to an initialized WC_RNG.

See:

- wc_MlKemKey_MakeKeyWithRandom
- wc_MlKemKey_Encapsulate

- wc_MlKemKey_Decapsulate

Return:

- 0 on success.
- BAD_FUNC_ARG if key or rng is NULL.
- MEMORY_E on allocation failure.

Example

```
MlKemKey key;
WC_RNG rng;

wc_MlKemKey_Init(&key, WC_ML_KEM_768, NULL, INVALID_DEVID);
wc_InitRng(&rng);

if (wc_MlKemKey_MakeKey(&key, &rng) != 0) {
    // error generating key pair
}
```

C.64.2.8 function wc_MlKemKey_MakeKeyWithRandom

```
int wc_MlKemKey_MakeKeyWithRandom(
    MlKemKey * key,
    const unsigned char * rand,
    int len
)
```

Deterministic key generation: produces an ML-KEM key pair from the supplied 64 bytes of randomness instead of an RNG. Useful for known-answer tests and for applications that derive key randomness from another secret.

Parameters:

- **key** Pointer to an initialized MlKemKey.
- **rand** Pointer to a buffer of randomness.
- **len** Length of rand in bytes; must be 64.

See: wc_MlKemKey_MakeKey

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL or len is not 64.

C.64.2.9 function wc_MlKemKey_CipherTextSize

```
int wc_MlKemKey_CipherTextSize(
    MlKemKey * key,
    word32 * len
)
```

Returns the ciphertext size in bytes for the variant selected on this key.

Parameters:

- **key** Pointer to an initialized MlKemKey.
- **len** Receives the ciphertext size in bytes.

See:

- wc_MlKemKey_SharedSecretSize

- wc_MlKemKey_Encapsulate

Return:

- 0 on success.
- BAD_FUNC_ARG if key or len is NULL.

C.64.2.10 function wc_MlKemKey_SharedSecretSize

```
int wc_MlKemKey_SharedSecretSize(
    MlKemKey * key,
    word32 * len
)
```

Returns the shared-secret size in bytes for ML-KEM. The value is the same (32 bytes) across all parameter sets but is queried programmatically for symmetry with wc_MlKemKey_CipherTextSize().

Parameters:

- **key** Pointer to an initialized MlKemKey.
- **len** Receives the shared-secret size in bytes.

See: wc_MlKemKey_CipherTextSize

Return:

- 0 on success.
- BAD_FUNC_ARG if key or len is NULL.

C.64.2.11 function wc_MlKemKey_Encapsulate

```
int wc_MlKemKey_Encapsulate(
    MlKemKey * key,
    unsigned char * ct,
    unsigned char * ss,
    WC_RNG * rng
)
```

Encapsulates a fresh shared secret against the public key held in key. Produces a ciphertext that the holder of the corresponding private key can pass to wc_MlKemKey_Decapsulate() to recover the same shared secret.

Parameters:

- **key** Pointer to an MlKemKey containing a public key.
- **ct** Buffer that receives the ciphertext.
- **ss** Buffer that receives the 32-byte shared secret.
- **rng** Pointer to an initialized WC_RNG.

See:

- wc_MlKemKey_EncapsulateWithRandom
- wc_MlKemKey_Decapsulate

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- BAD_STATE_E if the public key has not been set.
- NOT_COMPILED_IN if wolfSSL was built with WC_NO_RNG.
- MEMORY_E on allocation failure inside the encapsulation routine.

The ct buffer must be at least wc_MlKemKey_CipherTextSize() bytes and ss must be at least wc_MlKemKey_SharedSecretSize() bytes.

Example

```
MlKemKey key;
unsigned char ct[WC_ML_KEM_768_CIPHER_TEXT_SIZE];
unsigned char ss[WC_ML_KEM_SS_SZ];

// ... key holds the recipient's public key ...
if (wc_MlKemKey_Encapsulate(&key, ct, ss, &rng) != 0) {
    // error during encapsulation
}
// Send ct to the holder of the matching private key.
```

C.64.2.12 function wc_MlKemKey_EncapsulateWithRandom

```
int wc_MlKemKey_EncapsulateWithRandom(
    MlKemKey * key,
    unsigned char * ct,
    unsigned char * ss,
    const unsigned char * rand,
    int len
)
```

Deterministic variant of wc_MlKemKey_Encapsulate(). Uses the supplied 32 bytes of randomness instead of consuming output from an RNG.

Parameters:

- **key** Pointer to an MlKemKey containing a public key.
- **ct** Buffer that receives the ciphertext.
- **ss** Buffer that receives the 32-byte shared secret.
- **rand** Buffer of randomness.
- **len** Length of rand in bytes; must be 32.

See: wc_MlKemKey_Encapsulate

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL or len is not 32.

C.64.2.13 function wc_MlKemKey_Decapsulate

```
int wc_MlKemKey_Decapsulate(
    MlKemKey * key,
    unsigned char * ss,
    const unsigned char * ct,
    word32 len
)
```

Decapsulates a ciphertext using the private key held in key and recovers the shared secret produced by wc_MlKemKey_Encapsulate(). ML_KEM decapsulation is constant time and includes an implicit_rejection check on malformed ciphertexts (an attacker cannot learn the validity of ct from the runtime).

Parameters:

- **key** Pointer to an MlKemKey with the private key.

- **ss** Buffer that receives the 32-byte shared secret.
- **ct** The ciphertext to decapsulate.
- **len** Length of ct in bytes.

See:

- wc_MlKemKey_Encapsulate
- wc_MlKemKey_CipherTextSize

Return:

- 0 on success (a shared secret was written to ss).
- BAD_FUNC_ARG if any required pointer is NULL.
- BAD_STATE_E if the private key has not been set.
- BUFFER_E if len does not match the expected ciphertext size for the configured ML-KEM variant.
- NOT_COMPILED_IN if the key's ML-KEM variant was disabled at build time.
- MEMORY_E on allocation failure.

The ss buffer must be at least wc_MlKemKey_SharedSecretSize() bytes and ct must be exactly wc_MlKemKey_CipherTextSize() bytes.

C.64.2.14 function wc_MlKemKey_DecodePrivateKey

```
int wc_MlKemKey_DecodePrivateKey(
    MlKemKey * key,
    const unsigned char * in,
    word32 len
)
```

Decodes a raw ML-KEM private key into key. The variant must already be selected on the key (typically via wc_MlKemKey_Init() or wc_MlKemKey_New()) and len must match the private key size for that variant.

Parameters:

- **key** Pointer to an initialized MlKemKey.
- **in** Raw private key bytes.
- **len** Length of in in bytes.

See:

- wc_MlKemKey_EncodePrivateKey
- wc_MlKemKey_PrivateKeySize

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL or len does not match the expected size.

C.64.2.15 function wc_MlKemKey_DecodePublicKey

```
int wc_MlKemKey_DecodePublicKey(
    MlKemKey * key,
    const unsigned char * in,
    word32 len
)
```

Decodes a raw ML-KEM public key into key. The variant must already be selected on the key and len must match the public key size for that variant.

Parameters:

- **key** Pointer to an initialized MlKemKey.
- **in** Raw public key bytes.
- **len** Length of in in bytes.

See:

- wc_MlKemKey_EncodePublicKey
- wc_MlKemKey_PublicKeySize

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL or len does not match the expected size.

C.64.2.16 function wc_MlKemKey_PrivateKeySize

```
int wc_MlKemKey_PrivateKeySize(
    MlKemKey * key,
    word32 * len
)
```

Returns the encoded private key size in bytes for the variant selected on this key.

Parameters:

- **key** Pointer to an initialized MlKemKey.
- **len** Receives the private key size in bytes.

See: wc_MlKemKey_PublicKeySize

Return:

- 0 on success.
- BAD_FUNC_ARG if key or len is NULL.

C.64.2.17 function wc_MlKemKey_PublicKeySize

```
int wc_MlKemKey_PublicKeySize(
    MlKemKey * key,
    word32 * len
)
```

Returns the encoded public key size in bytes for the variant selected on this key.

Parameters:

- **key** Pointer to an initialized MlKemKey.
- **len** Receives the public key size in bytes.

See: wc_MlKemKey_PrivateKeySize

Return:

- 0 on success.
- BAD_FUNC_ARG if key or len is NULL.

C.64.2.18 function wc_MlKemKey_EncodePrivateKey

```
int wc_MlKemKey_EncodePrivateKey(
    MlKemKey * key,
    unsigned char * out,
```

```
    word32 len
)
```

Encodes the ML-KEM private key into out. The out buffer length must be exactly wc_MlKemKey_PrivateKeySize() bytes.

Parameters:

- **key** Pointer to an MlKemKey with a private key.
- **out** Buffer that receives the encoded private key.
- **len** Length of out in bytes.

See:

- wc_MlKemKey_DecodePrivateKey
- wc_MlKemKey_PrivateKeySize

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- BAD_STATE_E if the private and public keys are not both set on the key object.
- BUFFER_E if len does not exactly equal the encoded private key size for the configured ML-KEM variant.
- NOT_COMPILED_IN if the key's ML-KEM variant was disabled at build time.

C.64.2.19 function wc_MlKemKey_EncodePublicKey

```
int wc_MlKemKey_EncodePublicKey(
    MlKemKey * key,
    unsigned char * out,
    word32 len
)
```

Encodes the ML-KEM public key into out. The out buffer length must be exactly wc_MlKemKey_PublicKeySize() bytes.

Parameters:

- **key** Pointer to an MlKemKey with a public key.
- **out** Buffer that receives the encoded public key.
- **len** Length of out in bytes.

See:

- wc_MlKemKey_DecodePublicKey
- wc_MlKemKey_PublicKeySize

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- BAD_STATE_E if the public key has not been set.
- BUFFER_E if len does not exactly equal the encoded public key size for the configured ML-KEM variant.
- NOT_COMPILED_IN if the key's ML-KEM variant was disabled at build time.

C.64.2.20 function wc_MlKemKey_PublicKeyToDer

```
int wc_MlKemKey_PublicKeyToDer(
    MlKemKey * key,
```

```
    byte * output,
    word32 len,
    int withAlg
)
```

Encodes the ML-KEM public key as a DER SubjectPublicKeyInfo. Pass NULL for output to obtain the required length.

Parameters:

- **key** Pointer to an MlKemKey with a public key.
- **output** Buffer that receives the DER, or NULL for a length.
- **len** Length of output in bytes.
- **withAlg** Include the SubjectPublicKeyInfo wrapper when 1, or emit only the raw public key bytes when 0.

See:

- wc_MlKemKey_PublicKeyDecode
- wc_MlKemKey_PrivateKeyToDer

Return:

- Length of the encoding in bytes on success.
- BAD_FUNC_ARG if key is NULL or its type has no standardised OID.
- MEMORY_E if dynamic memory allocation fails.

C.64.2.21 function wc_MlKemKey_PrivateKeyToDer

```
int wc_MlKemKey_PrivateKeyToDer(
    MlKemKey * key,
    byte * output,
    word32 len
)
```

Encodes the ML-KEM private key as a DER PKCS#8 OneAsymmetricKey, carrying the expanded decapsulation key. Pass NULL for output to obtain the required length.

Parameters:

- **key** Pointer to an MlKemKey with a private key.
- **output** Buffer that receives the DER, or NULL for a length.
- **len** Length of output in bytes.

See:

- wc_MlKemKey_PrivateKeyDecode
- wc_MlKemKey_PublicKeyToDer

Return:

- Length of the encoding in bytes on success.
- BAD_FUNC_ARG if key is NULL or its type has no standardised OID.
- MEMORY_E if dynamic memory allocation fails.

C.64.2.22 function wc_MlKemKey_PublicKeyDecode

```
int wc_MlKemKey_PublicKeyDecode(
    MlKemKey * key,
    const byte * input,
    word32 inSz,
```

```
    word32 * inOutIdx
)
```

Decodes a DER SubjectPublicKeyInfo into an ML-KEM public key. Takes wrapped DER, where wc_MlKemKey_DecodePublicKey takes the raw encoded key. A key initialized for a parameter set holds the DER to that same one. A key initialized with WC_ML_KEM_TYPE_UNSET takes the parameter set from the algorithm OID in the DER instead.

Parameters:

- **key** Pointer to an initialized MlKemKey.
- **input** Buffer holding the DER.
- **inSz** Length of input in bytes.
- **inOutIdx** On in, index into input; on out, index after.

See:

- wc_MlKemKey_PublicKeyToDer
- wc_MlKemKey_Init

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- ASN_PARSE_E if the DER is invalid or names another parameter set.
- NOT_COMPILED_IN if the DER names a parameter set this build lacks.

C.64.2.23 function wc_MlKemKey_PrivateKeyDecode

```
int wc_MlKemKey_PrivateKeyDecode(
    MlKemKey * key,
    const byte * input,
    word32 inSz,
    word32 * inOutIdx
)
```

Decodes a DER PKCS#8 OneAsymmetricKey into an ML-KEM private key. Takes wrapped DER, where wc_MlKemKey_DecodePrivateKey takes the raw encoded key. All three RFC 9935 Section 6 CHOICE forms are accepted: the 64-byte seed under an implicit [0], the expanded decapsulation key as an OCTET STRING, and the SEQUENCE carrying both. A seed is expanded with ML-KEM.KeyGen_internal(d,z). For the “both” form the expanded key is regenerated from the seed and the two are compared, so a key whose halves disagree is rejected as malformed, as RFC 9935 Section 8 requires. A key initialized for a parameter set holds the DER to that same parameter set; a key initialized with WC_ML_KEM_TYPE_UNSET takes it from the algorithm OID in the DER instead.

Parameters:

- **key** Pointer to an initialized MlKemKey.
- **input** Buffer holding the DER.
- **inSz** Length of input in bytes.
- **inOutIdx** On in, index into input; on out, index after.

See: wc_MlKemKey_PrivateKeyToDer

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- MEMORY_E if dynamic memory allocation fails.
- ASN_PARSE_E if the DER is invalid, names another parameter set, carries a seed of the wrong length, or pairs a seed with an expanded key that does not match it.

- NOT_COMPILED_IN if the key carries a seed and the build defines WOLFSSL_MLKEM_NO_MAKE_KEY or WOLF_CRYPTO_CB_ONLY_MLKEM.

A build defining WOLFSSL_MLKEM_NO_MAKE_KEY or WOLF_CRYPTO_CB_ONLY_MLKEM cannot expand a seed, and so cannot perform the Section 8 comparison either. Such a build rejects every key that carries a seed, the "both" form included, rather than accepting its expanded half unchecked.

C.64.3 Source code

```
MlKemKey* wc_MlKemKey_New(int type, void* heap, int devId);

int wc_MlKemKey_Delete(MlKemKey* key, MlKemKey** key_p);

int wc_MlKemKey_Init(MlKemKey* key, int type, void* heap, int devId);

int wc_MlKemKey_Free(MlKemKey* key);

int wc_MlKemKey_Init_Id(MlKemKey* key, int type, const unsigned char* id,
    int len, void* heap, int devId);

int wc_MlKemKey_Init_Label(MlKemKey* key, int type, const char* label,
    void* heap, int devId);

int wc_MlKemKey_MakeKey(MlKemKey* key, WC_RNG* rng);

int wc_MlKemKey_MakeKeyWithRandom(MlKemKey* key, const unsigned char* rand,
    int len);

int wc_MlKemKey_CipherTextSize(MlKemKey* key, word32* len);

int wc_MlKemKey_SharedSecretSize(MlKemKey* key, word32* len);

int wc_MlKemKey_Encapsulate(MlKemKey* key, unsigned char* ct,
    unsigned char* ss, WC_RNG* rng);

int wc_MlKemKey_EncapsulateWithRandom(MlKemKey* key, unsigned char* ct,
    unsigned char* ss, const unsigned char* rand, int len);

int wc_MlKemKey_Decapsulate(MlKemKey* key, unsigned char* ss,
    const unsigned char* ct, word32 len);

int wc_MlKemKey_DecodePrivateKey(MlKemKey* key, const unsigned char* in,
    word32 len);

int wc_MlKemKey_DecodePublicKey(MlKemKey* key, const unsigned char* in,
    word32 len);

int wc_MlKemKey_PrivateKeySize(MlKemKey* key, word32* len);

int wc_MlKemKey_PublicKeySize(MlKemKey* key, word32* len);

int wc_MlKemKey_EncodePrivateKey(MlKemKey* key, unsigned char* out,
    word32 len);
```

```
int wc_MlKemKey_EncodePublicKey(MlKemKey* key, unsigned char* out,
    word32 len);

int wc_MlKemKey_PublicKeyToDer(MlKemKey* key, byte* output, word32 len,
    int withAlg);

int wc_MlKemKey_PrivateKeyToDer(MlKemKey* key, byte* output, word32 len);

int wc_MlKemKey_PublicKeyDecode(MlKemKey* key, const byte* input, word32 inSz,
    word32* inOutIdx);

int wc_MlKemKey_PrivateKeyDecode(MlKemKey* key, const byte* input, word32 inSz,
    word32* inOutIdx);
```

C.65 wc_port.h

C.65.1 Functions

	Name
int	**wolfCrypt_Init**(void)Used to initialize resources used by wolfCrypt.
int	**wolfCrypt_Cleanup**(void)Used to clean up resources used by wolfCrypt.
void	**wolfSSL_Atomic_Int_Init**(wolfSSL_Atomic_Int * c, int i)Initializes atomic integer.
void	**wolfSSL_Atomic_Uint_Init**(wolfSSL_Atomic_Uint * c, unsigned int i)Initializes atomic unsigned integer.
int	**wolfSSL_Atomic_Int_FetchAdd**(wolfSSL_Atomic_Int * c, int i)Atomically adds to integer and returns old value.
int	**wolfSSL_Atomic_Int_FetchSub**(wolfSSL_Atomic_Int * c, int i)Atomically subtracts from integer and returns old value.
int	**wolfSSL_Atomic_Int_AddFetch**(wolfSSL_Atomic_Int * c, int i)Atomically adds to integer and returns new value.
int	**wolfSSL_Atomic_Int_SubFetch**(wolfSSL_Atomic_Int * c, int i)Atomically subtracts from integer and returns new value.
int	**wolfSSL_Atomic_Int_CompareExchange**(wolfSSL_Atomic_Int * c, int * expected_i, int new_i)Atomically compares and exchanges integer.
unsigned int	**wolfSSL_Atomic_Uint_FetchAdd**(wolfSSL_Atomic_Uint * c, unsigned int i)Atomically adds to unsigned integer and returns old value.
unsigned int	**wolfSSL_Atomic_Uint_FetchSub**(wolfSSL_Atomic_Uint * c, unsigned int i)Atomically subtracts from unsigned integer, returns old value.
unsigned int	**wolfSSL_Atomic_Uint_AddFetch**(wolfSSL_Atomic_Uint * c, unsigned int i)Atomically adds to unsigned integer, returns new value.

	Name
unsigned int	**wolfSSL_Atomic_Uint_SubFetch**(wolfSSL_Atomic_Uint * c, unsigned int i)Atomically subtracts from unsigned integer, returns new value.
int	**wolfSSL_Atomic_Uint_CompareExchange**(wolfSSL_Atomic_Uint * c, unsigned int * expected_i, unsigned int new_i)Atomically compares and exchanges unsigned integer.
int	**wolfSSL_Atomic_Ptr_CompareExchange**(void ** c, void ** expected_ptr, void * new_ptr)Atomically compares and exchanges pointer.
int	**wc_InitMutex**(wolfSSL_Mutex * m)Initializes mutex.
int	**wc_FreeMutex**(wolfSSL_Mutex * m)Frees mutex resources.
int	**wc_LockMutex**(wolfSSL_Mutex * m)Locks mutex.
int	**wc_UnLockMutex**(wolfSSL_Mutex * m)Unlocks mutex.
wolfSSL_Mutex *	**wc_InitAndAllocMutex**(void)Initializes and allocates mutex.
int	**wc_InitRwLock**(wolfSSL_RwLock * m)Initializes read-write lock.
int	**wc_FreeRwLock**(wolfSSL_RwLock * m)Frees read-write lock resources.
int	**wc_LockRwLock_Wr**(wolfSSL_RwLock * m)Locks read-write lock for writing.
int	**wc_LockRwLock_Rd**(wolfSSL_RwLock * m)Locks read-write lock for reading.
int	**wc_UnLockRwLock**(wolfSSL_RwLock * m)Unlocks read-write lock.
int	**wc_LockMutex_ex**(int flag, int type, const char * file, int line)Locks mutex with debug info.
int	**wc_SetMutexCb**(mutex_cb * cb)Sets mutex callback.
mutex_cb *	**wc_GetMutexCb**(void)Gets mutex callback.
long	**wolfCrypt_heap_peakAllocs_checkpoint**(void)Checkpoints peak heap allocations.
long	**wolfCrypt_heap_peakBytes_checkpoint**(void)Checkpoints peak heap bytes.
int	**wc_FileLoad**(const char * fname, unsigned char ** buf, size_t * bufLen, void * heap)Loads file into buffer.
int	**wc_ReadDirFirst**(ReadDirCtx * ctx, const char * path, char ** name)Reads first entry in directory.
int	**wc_ReadDirNext**(ReadDirCtx * ctx, const char * path, char ** name)Reads next entry in directory.
void	**wc_ReadDirClose**(ReadDirCtx * ctx)Closes directory reading.

	Name
int	**wc_FileExists**(const char * fname)Checks if file exists.
int	**wolfSSL_GetHandleCbSet**(void)Checks if handle callback is set.
int	**wolfSSL_SetHandleCb**(wolfSSL_DSP_Handle_cb in)Sets handle callback.

C.65.2 Functions Documentation

C.65.2.1 function wolfCrypt_Init

```
int wolfCrypt_Init(
    void
)
```

Used to initialize resources used by wolfCrypt.

Parameters:

- **none** No parameters.

See: wolfCrypt_Cleanup

Return:

- 0 upon success.
- <0 upon failure of init resources.

Example

```
...
if (wolfCrypt_Init() != 0) {
    WOLFSSL_MSG("Error with wolfCrypt_Init call");
}
```

C.65.2.2 function wolfCrypt_Cleanup

```
int wolfCrypt_Cleanup(
    void
)
```

Used to clean up resources used by wolfCrypt.

Parameters:

- **none** No parameters.

See: wolfCrypt_Init

Return:

- 0 upon success.
- <0 upon failure of cleaning up resources.

Example

```
...
if (wolfCrypt_Cleanup() != 0) {
    WOLFSSL_MSG("Error with wolfCrypt_Cleanup call");
}
```

C.65.2.3 function wolfSSL_Atomic_Int_Init

```
void wolfSSL_Atomic_Int_Init(
    wolfSSL_Atomic_Int * c,
    int i
)
```

Initializes atomic integer.

Parameters:

- **c** Atomic integer pointer
- **i** Initial value

See: wolfSSL_Atomic_Int_FetchAdd

Return: none No returns

Example

```
wolfSSL_Atomic_Int counter;
wolfSSL_Atomic_Int_Init(&counter, 0);
```

C.65.2.4 function wolfSSL_Atomic_Uint_Init

```
void wolfSSL_Atomic_Uint_Init(
    wolfSSL_Atomic_Uint * c,
    unsigned int i
)
```

Initializes atomic unsigned integer.

Parameters:

- **c** Atomic unsigned integer pointer
- **i** Initial value

See: wolfSSL_Atomic_Uint_FetchAdd

Return: none No returns

Example

```
wolfSSL_Atomic_Uint counter;
wolfSSL_Atomic_Uint_Init(&counter, 0);
```

C.65.2.5 function wolfSSL_Atomic_Int_FetchAdd

```
int wolfSSL_Atomic_Int_FetchAdd(
    wolfSSL_Atomic_Int * c,
    int i
)
```

Atomically adds to integer and returns old value.

Parameters:

- **c** Atomic integer pointer
- **i** Value to add

See: wolfSSL_Atomic_Int_AddFetch

Return: Old value before addition

Example

```
wolfSSL_Atomic_Int counter;
int old = wolfSSL_Atomic_Int_FetchAdd(&counter, 1);
```

C.65.2.6 function wolfSSL_Atomic_Int_FetchSub

```
int wolfSSL_Atomic_Int_FetchSub(
    wolfSSL_Atomic_Int * c,
    int i
)
```

Atomically subtracts from integer and returns old value.

Parameters:

- **c** Atomic integer pointer
- **i** Value to subtract

See: wolfSSL_Atomic_Int_SubFetch

Return: Old value before subtraction

Example

```
wolfSSL_Atomic_Int counter;
int old = wolfSSL_Atomic_Int_FetchSub(&counter, 1);
```

C.65.2.7 function wolfSSL_Atomic_Int_AddFetch

```
int wolfSSL_Atomic_Int_AddFetch(
    wolfSSL_Atomic_Int * c,
    int i
)
```

Atomically adds to integer and returns new value.

Parameters:

- **c** Atomic integer pointer
- **i** Value to add

See: wolfSSL_Atomic_Int_FetchAdd

Return: New value after addition

Example

```
wolfSSL_Atomic_Int counter;
int new_val = wolfSSL_Atomic_Int_AddFetch(&counter, 1);
```

C.65.2.8 function wolfSSL_Atomic_Int_SubFetch

```
int wolfSSL_Atomic_Int_SubFetch(
    wolfSSL_Atomic_Int * c,
    int i
)
```

Atomically subtracts from integer and returns new value.

Parameters:

- **c** Atomic integer pointer
- **i** Value to subtract

See: wolfSSL_Atomic_Int_FetchSub

Return: New value after subtraction

Example

```
wolfSSL_Atomic_Int counter;
int new_val = wolfSSL_Atomic_Int_SubFetch(&counter, 1);
```

C.65.2.9 function wolfSSL_Atomic_Int_CompareExchange

```
int wolfSSL_Atomic_Int_CompareExchange(
    wolfSSL_Atomic_Int * c,
    int * expected_i,
    int new_i
)
```

Atomically compares and exchanges integer.

Parameters:

- **c** Atomic integer pointer
- **expected_i** Pointer to expected value
- **new_i** New value to set

See: wolfSSL_Atomic_Int_FetchAdd

Return: 1 if exchange occurred, 0 otherwise

Example

```
wolfSSL_Atomic_Int counter;
int expected = 0;
int ret = wolfSSL_Atomic_Int_CompareExchange(&counter, &expected, 1);
```

C.65.2.10 function wolfSSL_Atomic_Uint_FetchAdd

```
unsigned int wolfSSL_Atomic_Uint_FetchAdd(
    wolfSSL_Atomic_Uint * c,
    unsigned int i
)
```

Atomically adds to unsigned integer and returns old value.

Parameters:

- **c** Atomic unsigned integer pointer
- **i** Value to add

See: wolfSSL_Atomic_Uint_AddFetch

Return: Old value before addition

Example

```
wolfSSL_Atomic_Uint counter;
unsigned int old = wolfSSL_Atomic_Uint_FetchAdd(&counter, 1);
```

C.65.2.11 function wolfSSL_Atomic_Uint_FetchSub

```
unsigned int wolfSSL_Atomic_Uint_FetchSub(
    wolfSSL_Atomic_Uint * c,
    unsigned int i
)
```

Atomically subtracts from unsigned integer, returns old value.

Parameters:

- **c** Atomic unsigned integer pointer
- **i** Value to subtract

See: wolfSSL_Atomic_Uint_SubFetch

Return: Old value before subtraction

Example

```
wolfSSL_Atomic_Uint counter;
unsigned int old = wolfSSL_Atomic_Uint_FetchSub(&counter, 1);
```

C.65.2.12 function wolfSSL_Atomic_Uint_AddFetch

```
unsigned int wolfSSL_Atomic_Uint_AddFetch(
    wolfSSL_Atomic_Uint * c,
    unsigned int i
)
```

Atomically adds to unsigned integer, returns new value.

Parameters:

- **c** Atomic unsigned integer pointer
- **i** Value to add

See: wolfSSL_Atomic_Uint_FetchAdd

Return: New value after addition

Example

```
wolfSSL_Atomic_Uint counter;
unsigned int new_val = wolfSSL_Atomic_Uint_AddFetch(&counter, 1);
```

C.65.2.13 function wolfSSL_Atomic_Uint_SubFetch

```
unsigned int wolfSSL_Atomic_Uint_SubFetch(
    wolfSSL_Atomic_Uint * c,
    unsigned int i
)
```

Atomically subtracts from unsigned integer, returns new value.

Parameters:

- **c** Atomic unsigned integer pointer
- **i** Value to subtract

See: wolfSSL_Atomic_Uint_FetchSub

Return: New value after subtraction

Example

```
wolfSSL_Atomic_Uint counter;
unsigned int new_val = wolfSSL_Atomic_Uint_SubFetch(&counter, 1);
```

C.65.2.14 function wolfSSL_Atomic_Uint_CompareExchange

```
int wolfSSL_Atomic_Uint_CompareExchange(
    wolfSSL_Atomic_Uint * c,
    unsigned int * expected_i,
    unsigned int new_i
)
```

Atomically compares and exchanges unsigned integer.

Parameters:

- **c** Atomic unsigned integer pointer
- **expected_i** Pointer to expected value
- **new_i** New value to set

See: wolfSSL_Atomic_Uint_FetchAdd

Return: 1 if exchange occurred, 0 otherwise

Example

```
wolfSSL_Atomic_Uint counter;
unsigned int expected = 0;
int ret = wolfSSL_Atomic_Uint_CompareExchange(&counter, &expected, 1);
```

C.65.2.15 function wolfSSL_Atomic_Ptr_CompareExchange

```
int wolfSSL_Atomic_Ptr_CompareExchange(
    void ** c,
    void ** expected_ptr,
    void * new_ptr
)
```

Atomically compares and exchanges pointer.

Parameters:

- **c** Pointer to pointer
- **expected_ptr** Pointer to expected pointer value
- **new_ptr** New pointer value

See: wolfSSL_Atomic_Int_CompareExchange

Return: 1 if exchange occurred, 0 otherwise

Example

```
void* ptr = NULL;
void* expected = NULL;
void* new_val = malloc(100);
int ret = wolfSSL_Atomic_Ptr_CompareExchange(&ptr, &expected, new_val);
```

C.65.2.16 function wc_InitMutex

```
int wc_InitMutex(
    wolfSSL_Mutex * m
)
```

Initializes mutex.

Parameters:

- **m** Mutex pointer

See: wc_FreeMutex

Return:

- 0 on success
- negative on error

Example

```
wolfSSL_Mutex mutex;
int ret = wc_InitMutex(&mutex);
```

C.65.2.17 function wc_FreeMutex

```
int wc_FreeMutex(
    wolfSSL_Mutex * m
)
```

Frees mutex resources.

Parameters:

- **m** Mutex pointer

See: wc_InitMutex

Return:

- 0 on success
- negative on error

Example

```
wolfSSL_Mutex mutex;
wc_InitMutex(&mutex);
int ret = wc_FreeMutex(&mutex);
```

C.65.2.18 function wc_LockMutex

```
int wc_LockMutex(
    wolfSSL_Mutex * m
)
```

Locks mutex.

Parameters:

- **m** Mutex pointer

See: wc_UnLockMutex

Return:

- 0 on success
- negative on error

Example

```
wolfSSL_Mutex mutex;
int ret = wc_LockMutex(&mutex);
```

C.65.2.19 function wc_UnLockMutex

```
int wc_UnLockMutex(
    wolfSSL_Mutex * m
)
```

Unlocks mutex.

Parameters:

- **m** Mutex pointer

See: wc_LockMutex

Return:

- 0 on success
- negative on error

Example

```
wolfSSL_Mutex mutex;
wc_LockMutex(&mutex);
int ret = wc_UnLockMutex(&mutex);
```

C.65.2.20 function wc_InitAndAllocMutex

```
wolfSSL_Mutex * wc_InitAndAllocMutex(
    void
)
```

Initializes and allocates mutex.

Parameters:

- **none** No parameters

See: wc_InitMutex

Return:

- Pointer to mutex on success
- NULL on error

Example

```
wolfSSL_Mutex* mutex = wc_InitAndAllocMutex();
if (mutex != NULL) {
    wc_LockMutex(mutex);
}
```

C.65.2.21 function wc_InitRwLock

```
int wc_InitRwLock(
    wolfSSL_RwLock * m
)
```

Initializes read-write lock.

Parameters:

- **m** Read-write lock pointer

See: wc_FreeRwLock

Return:

- 0 on success
- negative on error

Example

```
wolfSSL_RwLock lock;
int ret = wc_InitRwLock(&lock);
```

C.65.2.22 function wc_FreeRwLock

```
int wc_FreeRwLock(
    wolfSSL_RwLock * m
)
```

Frees read-write lock resources.

Parameters:

- **m** Read-write lock pointer

See: wc_InitRwLock

Return:

- 0 on success
- negative on error

Example

```
wolfSSL_RwLock lock;
wc_InitRwLock(&lock);
int ret = wc_FreeRwLock(&lock);
```

C.65.2.23 function wc_LockRwLock_Wr

```
int wc_LockRwLock_Wr(
    wolfSSL_RwLock * m
)
```

Locks read-write lock for writing.

Parameters:

- **m** Read-write lock pointer

See: wc_UnLockRwLock

Return:

- 0 on success
- negative on error

Example

```
wolfSSL_RwLock lock;
int ret = wc_LockRwLock_Wr(&lock);
```

C.65.2.24 function wc_LockRwLock_Rd

```
int wc_LockRwLock_Rd(
    wolfSSL_RwLock * m
)
```

Locks read-write lock for reading.

Parameters:

- **m** Read-write lock pointer

See: wc_UnLockRwLock

Return:

- 0 on success
- negative on error

Example

```
wolfSSL_RwLock lock;
int ret = wc_LockRwLock_Rd(&lock);
```

C.65.2.25 function wc_UnLockRwLock

```
int wc_UnLockRwLock(
    wolfSSL_RwLock * m
)
```

Unlocks read-write lock.

Parameters:

- **m** Read-write lock pointer

See: wc_LockRwLock_Rd

Return:

- 0 on success
- negative on error

Example

```
wolfSSL_RwLock lock;
wc_LockRwLock_Rd(&lock);
int ret = wc_UnLockRwLock(&lock);
```

C.65.2.26 function wc_LockMutex_ex

```
int wc_LockMutex_ex(
    int flag,
    int type,
    const char * file,
    int line
)
```

Locks mutex with debug info.

Parameters:

- **flag** Lock flag
- **type** Lock type

- **file** Source file name
- **line** Source line number

See: wc_LockMutex

Return:

- 0 on success
- negative on error

Example

```
int ret = wc_LockMutex_ex(0, 0, __FILE__, __LINE__);
```

C.65.2.27 function wc_SetMutexCb

```
int wc_SetMutexCb(
    mutex_cb * cb
)
```

Sets mutex callback.

Parameters:

- **cb** Mutex callback pointer

See: wc_GetMutexCb

Return:

- 0 on success
- negative on error

Example

```
mutex_cb cb;
int ret = wc_SetMutexCb(&cb);
```

C.65.2.28 function wc_GetMutexCb

```
mutex_cb * wc_GetMutexCb(
    void
)
```

Gets mutex callback.

Parameters:

- **none** No parameters

See: wc_SetMutexCb

Return: Pointer to mutex callback

Example

```
mutex_cb* cb = wc_GetMutexCb();
```

C.65.2.29 function wolfCrypt_heap_peakAllocs_checkpoint

```
long wolfCrypt_heap_peakAllocs_checkpoint(
    void
)
```

Checkpoints peak heap allocations.

Parameters:

- **none** No parameters

See: wolfCrypt_heap_peakBytes_checkpoint

Return: Peak allocation count

Example

```
long peak = wolfCrypt_heap_peakAllocs_checkpoint();
```

C.65.2.30 function wolfCrypt_heap_peakBytes_checkpoint

```
long wolfCrypt_heap_peakBytes_checkpoint(
    void
)
```

Checkpoints peak heap bytes.

Parameters:

- **none** No parameters

See: wolfCrypt_heap_peakAllocs_checkpoint

Return: Peak bytes allocated

Example

```
long peak = wolfCrypt_heap_peakBytes_checkpoint();
```

C.65.2.31 function wc_FileLoad

```
int wc_FileLoad(
    const char * fname,
    unsigned char ** buf,
    size_t * bufLen,
    void * heap
)
```

Loads file into buffer.

Parameters:

- **fname** File name
- **buf** Buffer pointer
- **bufLen** Buffer length pointer
- **heap** Heap hint

See: wc_FileExists

Return:

- 0 on success
- negative on error

Example

```
unsigned char* buf = NULL;
size_t len = 0;
int ret = wc_FileLoad("file.txt", &buf, &len, NULL);
```

C.65.2.32 function wc_ReadDirFirst

```
int wc_ReadDirFirst(
    ReadDirCtx * ctx,
    const char * path,
    char ** name
)
```

Reads first entry in directory.

Parameters:

- **ctx** Directory context
- **path** Directory path
- **name** Pointer to store entry name

See: wc_ReadDirNext

Return:

- 0 on success
- negative on error

Example

```
ReadDirCtx ctx;
char* name;
int ret = wc_ReadDirFirst(&ctx, "/path", &name);
```

C.65.2.33 function wc_ReadDirNext

```
int wc_ReadDirNext(
    ReadDirCtx * ctx,
    const char * path,
    char ** name
)
```

Reads next entry in directory.

Parameters:

- **ctx** Directory context
- **path** Directory path
- **name** Pointer to store entry name

See: wc_ReadDirFirst

Return:

- 0 on success
- negative on error

Example

```
ReadDirCtx ctx;
char* name;
int ret = wc_ReadDirNext(&ctx, "/path", &name);
```

C.65.2.34 function wc_ReadDirClose

```
void wc_ReadDirClose(
    ReadDirCtx * ctx
)
```

Closes directory reading.

Parameters:

- **ctx** Directory context

See: wc_ReadDirFirst

Return: none No returns

Example

```
ReadDirCtx ctx;
wc_ReadDirClose(&ctx);
```

C.65.2.35 function wc_FileExists

```
int wc_FileExists(
    const char * fname
)
```

Checks if file exists.

Parameters:

- **fname** File name

See: wc_FileLoad

Return:

- 1 if file exists
- 0 if file does not exist

Example

```
if (wc_FileExists("file.txt")) {
    // file exists
}
```

C.65.2.36 function wolfSSL_GetHandleCbSet

```
int wolfSSL_GetHandleCbSet(
    void
)
```

Checks if handle callback is set.

Parameters:

- **none** No parameters

See: wolfSSL_SetHandleCb

Return:

- 1 if set
- 0 if not set

Example

```
if (wolfSSL_GetHandleCbSet()) {
    // callback is set
}
```

C.65.2.37 function wolfSSL_SetHandleCb

```
int wolfSSL_SetHandleCb(
    wolfSSL_DSP_Handle_cb in
)
```

Sets handle callback.

Parameters:

- **in** Handle callback

See: wolfSSL_GetHandleCbSet

Return:

- 0 on success
- negative on error

Example

```
int ret = wolfSSL_SetHandleCb(myHandleCallback);
```

C.65.3 Source code

```
int wolfCrypt_Init(void);

int wolfCrypt_Cleanup(void);

void wolfSSL_Atomic_Int_Init(wolfSSL_Atomic_Int* c, int i);

void wolfSSL_Atomic_Uint_Init(wolfSSL_Atomic_Uint* c, unsigned int i);

int wolfSSL_Atomic_Int_FetchAdd(wolfSSL_Atomic_Int* c, int i);

int wolfSSL_Atomic_Int_FetchSub(wolfSSL_Atomic_Int* c, int i);

int wolfSSL_Atomic_Int_AddFetch(wolfSSL_Atomic_Int* c, int i);

int wolfSSL_Atomic_Int_SubFetch(wolfSSL_Atomic_Int* c, int i);

int wolfSSL_Atomic_Int_CompareExchange(wolfSSL_Atomic_Int* c,
                                       int *expected_i, int new_i);

unsigned int wolfSSL_Atomic_Uint_FetchAdd(wolfSSL_Atomic_Uint* c,
                                          unsigned int i);

unsigned int wolfSSL_Atomic_Uint_FetchSub(wolfSSL_Atomic_Uint* c,
                                          unsigned int i);

unsigned int wolfSSL_Atomic_Uint_AddFetch(wolfSSL_Atomic_Uint* c,
                                          unsigned int i);

unsigned int wolfSSL_Atomic_Uint_SubFetch(wolfSSL_Atomic_Uint* c,
                                          unsigned int i);

int wolfSSL_Atomic_Uint_CompareExchange(wolfSSL_Atomic_Uint* c,
```

```
                                        unsigned int *expected_i,
                                        unsigned int new_i);

int wolfSSL_Atomic_Ptr_CompareExchange(void** c, void **expected_ptr,
                                       void *new_ptr);

int wc_InitMutex(wolfSSL_Mutex* m);

int wc_FreeMutex(wolfSSL_Mutex* m);

int wc_LockMutex(wolfSSL_Mutex* m);

int wc_UnLockMutex(wolfSSL_Mutex* m);

wolfSSL_Mutex* wc_InitAndAllocMutex(void);

int wc_InitRwLock(wolfSSL_RwLock* m);

int wc_FreeRwLock(wolfSSL_RwLock* m);

int wc_LockRwLock_Wr(wolfSSL_RwLock* m);

int wc_LockRwLock_Rd(wolfSSL_RwLock* m);

int wc_UnLockRwLock(wolfSSL_RwLock* m);

int wc_LockMutex_ex(int flag, int type, const char* file, int line);

int wc_SetMutexCb(mutex_cb* cb);

mutex_cb* wc_GetMutexCb(void);

long wolfCrypt_heap_peakAllocs_checkpoint(void);

long wolfCrypt_heap_peakBytes_checkpoint(void);

int wc_FileLoad(const char* fname, unsigned char** buf, size_t* bufLen,
                void* heap);

int wc_ReadDirFirst(ReadDirCtx* ctx, const char* path, char** name);

int wc_ReadDirNext(ReadDirCtx* ctx, const char* path, char** name);

void wc_ReadDirClose(ReadDirCtx* ctx);

int wc_FileExists(const char* fname);

int wolfSSL_GetHandleCbSet(void);

int wolfSSL_SetHandleCb(wolfSSL_DSP_Handle_cb in);
```

C.66 wc_she.h

C.66.1 Functions

	Name
int	**wc_SHE_Init**(wc_SHE * she, void * heap, int devId)Initialize a SHE context with a heap hint and device ID.
int	**wc_SHE_Init_Id**(wc_SHE * she, unsigned char * id, int len, void * heap, int devId)Initialize a SHE context with an opaque hardware key identifier. Useful when using crypto callbacks and additional info needs to be attached to the SHE context to determine slot or key group information.
int	**wc_SHE_Init_Label**(wc_SHE * she, const char * label, void * heap, int devId)Initialize a SHE context with a human-readable key label. Useful when using crypto callbacks and additional info needs to be attached to the SHE context to determine slot or key group information.
void	**wc_SHE_Free**(wc_SHE * she)Scrub all data and zero the SHE context. Safe to call on a NULL pointer.
int	**wc_SHE_GetUID**(wc_SHE * she, byte * uid, word32 uidSz, const void * ctx)Retrieve the UID from hardware via a crypto callback. Requires WOLF_CRYPTO_CB and that NO_WC_SHE_GETUID is not defined.
int	**wc_SHE_GetCounter**(wc_SHE * she, word32 * counter, const void * ctx)Retrieve the monotonic counter value from hardware via a crypto callback. The SHE spec uses a 28-bit counter. The caller should increment this value before passing to GenerateM1M2M3 or GenerateM4M5. Requires WOLF_CRYPTO_CB and that NO_WC_SHE_GETCOUNTER is not defined.
int	**wc_SHE_SetKdfConstants**(wc_SHE * she, const byte * encC, word32 encCSz, const byte * macC, word32 macCSz)Set custom KDF constants used in Miyaguchi-Preneel key derivation. Defaults are KEY_UPDATE_ENC_C and KEY_UPDATE_MAC_C from the SHE spec. Either pointer may be NULL to leave that constant unchanged. Requires WOLFSSL_SHE_EXTENDED.

	Name
int	**wc_SHE_SetM2Header**(wc_SHE * she, const byte * header, word32 headerSz)Override the M2 cleartext header (first 16 bytes of M2 before encryption). When set, GenerateM1M2M3 uses this instead of auto-building from counter and flags. Requires WOLFSSL_SHE_EXTENDED.
int	**wc_SHE_SetM4Header**(wc_SHE * she, const byte * header, word32 headerSz)Override the M4 cleartext counter block (16_byte block encrypted with K3). When set, GenerateM4M5 uses this instead of auto-building from counter. Requires WOLFSSL_SHE_EXTENDED.
int	**wc_SHE_ImportM1M2M3**(wc_SHE * she, const byte * m1, word32 m1Sz, const byte * m2, word32 m2Sz, const byte * m3, word32 m3Sz)Import externally-provided M1/M2/M3 into the SHE context. Sets the generated flag so the callback for GenerateM4M5 can read M1/M2/M3 from the context to send to hardware. Requires WOLF_CRYPTO_CB and that NO_WC_SHE_IMPORT_M123 is not defined.
int	**wc_SHE_GenerateM1M2M3**(wc_SHE * she, const byte * uid, word32 uidSz, byte authKeyId, const byte * authKey, word32 authKeySz, byte targetKeyId, const byte * newKey, word32 newKeySz, word32 counter, byte flags, byte * m1, word32 m1Sz, byte * m2, word32 m2Sz, byte * m3, word32 m3Sz)Generate SHE key update messages M1, M2, and M3 and write them to caller_provided buffers. Uses Miyaguchi_Preneel AES_128 KDF to derive K1 and K2 from the authorizing key, AES_CBC to encrypt the new key (M2), and AES_CMAC for authentication (M3).
int	**wc_SHE_GenerateM4M5**(wc_SHE * she, const byte * uid, word32 uidSz, byte authKeyId, byte targetKeyId, const byte * newKey, word32 newKeySz, word32 counter, byte * m4, word32 m4Sz, byte * m5, word32 m5Sz)Generate SHE verification messages M4 and M5 and write them to caller-provided buffers. Uses Miyaguchi-Preneel AES-128 KDF to derive K3 and K4 from the new key, AES-ECB for the M4 counter block, and AES-CMAC for M5. Independent of M1/M2/M3 and can be called on a separate context.

COPYRIGHT ©2024 wolfSSL Inc.

	Name
int	**wc_SHE_LoadKey**(void * heap, int devId, const byte * m1, word32 m1Sz, const byte * m2, word32 m2Sz, const byte * m3, word32 m3Sz, byte * m4, word32 m4Sz, byte * m5, word32 m5Sz)One_shot convenience wrapper: Init, ImportM1M2M3, GenerateM4M5 (via callback), Free. Dispatches to a hardware crypto callback that sends M1/M2/M3 to the HSM and returns M4/M5. Requires a valid devId (not INVALID_DEVID). Define NO_WC_SHE_LOADKEY to compile out.
int	**wc_SHE_LoadKey_Id**(unsigned char * id, int idLen, void * heap, int devId, const byte * m1, word32 m1Sz, const byte * m2, word32 m2Sz, const byte * m3, word32 m3Sz, byte * m4, word32 m4Sz, byte * m5, word32 m5Sz)One-shot Load Key with an opaque hardware key identifier. Same as wc_SHE_LoadKey but initializes the context with wc_SHE_Init_Id. Define NO_WC_SHE_LOADKEY to compile out.
int	**wc_SHE_LoadKey_Label**(const char * label, void * heap, int devId, const byte * m1, word32 m1Sz, const byte * m2, word32 m2Sz, const byte * m3, word32 m3Sz, byte * m4, word32 m4Sz, byte * m5, word32 m5Sz)One-shot Load Key with a human-readable key label. Same as wc_SHE_LoadKey but initializes the context with wc_SHE_Init_Label. Define NO_WC_SHE_LOADKEY to compile out.
int	**wc_SHE_LoadKey_Verify**(void * heap, int devId, const byte * m1, word32 m1Sz, const byte * m2, word32 m2Sz, const byte * m3, word32 m3Sz, byte * m4, word32 m4Sz, byte * m5, word32 m5Sz, const byte * m4Expected, word32 m4ExpectedSz, const byte * m5Expected, word32 m5ExpectedSz)One-shot Load Key with M4/M5 verification. Same as wc_SHE_LoadKey but also compares the M4/M5 returned by the HSM against caller-provided expected values using constant-time comparison. Returns SIG_VERIFY_E on mismatch. The actual M4/M5 are still written to the output buffers on failure. Define NO_WC_SHE_LOADKEY to compile out.

	Name
int	**wc_SHE_LoadKey_Verify_Id**(unsigned char * id, int idLen, void * heap, int devId, const byte * m1, word32 m1Sz, const byte * m2, word32 m2Sz, const byte * m3, word32 m3Sz, byte * m4, word32 m4Sz, byte * m5, word32 m5Sz, const byte * m4Expected, word32 m4ExpectedSz, const byte * m5Expected, word32 m5ExpectedSz)One-shot Load Key with opaque key identifier and M4/M5 verification. Combines wc_SHE_LoadKey_Id and wc_SHE_LoadKey_Verify. Define NO_WC_SHE_LOADKEY to compile out.
int	**wc_SHE_LoadKey_Verify_Label**(const char * label, void * heap, int devId, const byte * m1, word32 m1Sz, const byte * m2, word32 m2Sz, const byte * m3, word32 m3Sz, byte * m4, word32 m4Sz, byte * m5, word32 m5Sz, const byte * m4Expected, word32 m4ExpectedSz, const byte * m5Expected, word32 m5ExpectedSz)One-shot Load Key with key label and M4/M5 verification. Combines wc_SHE_LoadKey_Label and wc_SHE_LoadKey_Verify. Define NO_WC_SHE_LOADKEY to compile out.
int	**wc_SHE_ExportKey**(wc_SHE * she, byte * m1, word32 m1Sz, byte * m2, word32 m2Sz, byte * m3, word32 m3Sz, byte * m4, word32 m4Sz, byte * m5, word32 m5Sz, const void * ctx)Export a key from hardware in SHE loadable format (M1_M5). Some HSMs allow exporting certain key slots (e.g. RAM key) so they can be re-loaded later via the SHE key update protocol. Requires WOLF_CRYPTO_CB and that NO_WC_SHE_EXPORTKEY is not defined. Any output buffer may be NULL to skip that message.
int	**wc_SHE_AesMp16**(Aes * aes, const byte * in, word32 inSz, byte * out)Miyaguchi_Preneel AES_128 one_way compression function. H_0 = 0, H_i = E_{H_{i_1}}(M_i) XOR M_i XOR H_{i-1}. Only valid for AES-128 where key size equals block size. This is an internal function exposed for testing purposes.

C.66.2 Functions Documentation

C.66.2.1 function wc_SHE_Init

```
int wc_SHE_Init(
    wc_SHE * she,
    void * heap,
    int devId
```

)

Initialize a SHE context with a heap hint and device ID.

Parameters:

- **she** pointer to a wc_SHE structure to initialize
- **heap** heap hint for internal allocations, or NULL
- **devId** crypto callback device ID, or INVALID_DEVID for software-only operation

See:

- wc_SHE_Init_Id
- wc_SHE_Init_Label
- wc_SHE_Free

Return:

- 0 on success
- BAD_FUNC_ARG if she is NULL

Example

```
wc_SHE she;
int ret;
ret = wc_SHE_Init(&she, NULL, INVALID_DEVID);
if (ret == 0) {
    // use she context
}
wc_SHE_Free(&she);
```

C.66.2.2 function wc_SHE_Init_Id

```
int wc_SHE_Init_Id(
    wc_SHE * she,
    unsigned char * id,
    int len,
    void * heap,
    int devId
)
```

Initialize a SHE context with an opaque hardware key identifier. Useful when using crypto callbacks and additional info needs to be attached to the SHE context to determine slot or key group information.

Parameters:

- **she** pointer to a wc_SHE structure to initialize
- **id** opaque key identifier bytes
- **len** length of id in bytes (0 to WC_SHE_MAX_ID_LEN)
- **heap** heap hint for internal allocations, or NULL
- **devId** crypto callback device ID

See:

- wc_SHE_Init
- wc_SHE_Init_Label
- wc_SHE_Free

Return:

- 0 on success
- BAD_FUNC_ARG if she is NULL, id is NULL when len > 0, or len exceeds WC_SHE_MAX_ID_LEN

Example

```
wc_SHE she;
unsigned char myId[] = { 0x01, 0x02, 0x03 };
int ret;
ret = wc_SHE_Init_Id(&she, myId, sizeof(myId), NULL, myDevId);
```

C.66.2.3 function wc_SHE_Init_Label

```
int wc_SHE_Init_Label(
    wc_SHE * she,
    const char * label,
    void * heap,
    int devId
)
```

Initialize a SHE context with a human-readable key label. Useful when using crypto callbacks and additional info needs to be attached to the SHE context to determine slot or key group information.

Parameters:

- **she** pointer to a wc_SHE structure to initialize
- **label** NUL-terminated key label string
- **heap** heap hint for internal allocations, or NULL
- **devId** crypto callback device ID

See:

- wc_SHE_Init
- wc_SHE_Init_Id
- wc_SHE_Free

Return:

- 0 on success
- BAD_FUNC_ARG if she or label is NULL, or label length exceeds WC_SHE_MAX_LABEL_LEN

Example

```
wc_SHE she;
int ret;
ret = wc_SHE_Init_Label(&she, "ecu-master-key", NULL, myDevId);
```

C.66.2.4 function wc_SHE_Free

```
void wc_SHE_Free(
    wc_SHE * she
)
```

Scrub all data and zero the SHE context. Safe to call on a NULL pointer.

Parameters:

- **she** pointer to a wc_SHE structure, or NULL

See: wc_SHE_Init

Example

```
wc_SHE she;
wc_SHE_Init(&she, NULL, INVALID_DEVID);
```

```
// ... use context ...
wc_SHE_Free(&she);
```

C.66.2.5 function wc_SHE_GetUID

```
int wc_SHE_GetUID(
    wc_SHE * she,
    byte * uid,
    word32 uidSz,
    const void * ctx
)
```

Retrieve the UID from hardware via a crypto callback. Requires WOLF_CRYPTO_CB and that NO_WC_SHE_GETUID is not defined.

Parameters:

- **she** initialized SHE context
- **uid** buffer to receive the 15-byte (120-bit) SHE UID
- **uidSz** size of uid buffer in bytes (must be >= WC_SHE_UID_SZ)
- **ctx** read-only caller context passed to the callback (e.g. challenge buffer, HSM handle)

See: wc_SHE_GetCounter

Return:

- 0 on success
- BAD_FUNC_ARG if she, uid, or uidSz is invalid
- CRYPTOCB_UNAVAILABLE if no callback is registered

Example

```
byte uid[WC_SHE_UID_SZ];
int ret;
ret = wc_SHE_GetUID(&she, uid, sizeof(uid), NULL);
```

C.66.2.6 function wc_SHE_GetCounter

```
int wc_SHE_GetCounter(
    wc_SHE * she,
    word32 * counter,
    const void * ctx
)
```

Retrieve the monotonic counter value from hardware via a crypto callback. The SHE spec uses a 28-bit counter. The caller should increment this value before passing to GenerateM1M2M3 or GenerateM4M5. Requires WOLF_CRYPTO_CB and that NO_WC_SHE_GETCOUNTER is not defined.

Parameters:

- **she** initialized SHE context
- **counter** pointer to receive the current counter value
- **ctx** read-only caller context passed to the callback

See: wc_SHE_GetUID

Return:

- 0 on success
- BAD_FUNC_ARG if she or counter is NULL
- CRYPTOCB_UNAVAILABLE if no callback is registered

Example

```
word32 counter;
int ret;
ret = wc_SHE_GetCounter(&she, &counter, NULL);
```

C.66.2.7 function wc_SHE_SetKdfConstants

```
int wc_SHE_SetKdfConstants(
    wc_SHE * she,
    const byte * encC,
    word32 encCSz,
    const byte * macC,
    word32 macCSz
)
```

Set custom KDF constants used in Miyaguchi-Preneel key derivation. Defaults are KEY_UPDATE_ENC_C and KEY_UPDATE_MAC_C from the SHE spec. Either pointer may be NULL to leave that constant unchanged. Requires WOLFSSL_SHE_EXTENDED.

Parameters:

- **she** initialized SHE context
- **encC** 16-byte encryption derivation constant (CENC), or NULL
- **encCSz** must be WC_SHE_KEY_SZ (16) when encC is non-NULL
- **macC** 16-byte MAC derivation constant (CMAC), or NULL
- **macCSz** must be WC_SHE_KEY_SZ (16) when macC is non-NULL

See:

- wc_SHE_SetM2Header
- wc_SHE_SetM4Header
- wc_SHE_GenerateM1M2M3

Return:

- 0 on success
- BAD_FUNC_ARG if she is NULL or sizes are not WC_SHE_KEY_SZ when corresponding pointer is non-NULL

Example

```
byte myEncC[WC_SHE_KEY_SZ] = { ... };
byte myMacC[WC_SHE_KEY_SZ] = { ... };
int ret;
ret = wc_SHE_SetKdfConstants(&she, myEncC, WC_SHE_KEY_SZ,
                             myMacC, WC_SHE_KEY_SZ);
```

C.66.2.8 function wc_SHE_SetM2Header

```
int wc_SHE_SetM2Header(
    wc_SHE * she,
    const byte * header,
    word32 headerSz
)
```

Override the M2 cleartext header (first 16 bytes of M2 before encryption). When set, GenerateM1M2M3 uses this instead of auto-building from counter and flags. Requires WOLFSSL_SHE_EXTENDED.

Parameters:

- **she** initialized SHE context
- **header** 16-byte cleartext header block
- **headerSz** must be WC_SHE_KEY_SZ (16)

See:

- wc_SHE_SetKdfConstants
- wc_SHE_SetM4Header
- wc_SHE_GenerateM1M2M3

Return:

- 0 on success
- BAD_FUNC_ARG if she or header is NULL, or headerSz is not WC_SHE_KEY_SZ

Example

```
byte header[WC_SHE_KEY_SZ] = { ... };
int ret;
ret = wc_SHE_SetM2Header(&she, header, WC_SHE_KEY_SZ);
```

C.66.2.9 function wc_SHE_SetM4Header

```
int wc_SHE_SetM4Header(
    wc_SHE * she,
    const byte * header,
    word32 headerSz
)
```

Override the M4 cleartext counter block (16-byte block encrypted with K3). When set, GenerateM4M5 uses this instead of auto-building from counter. Requires WOLFSSL_SHE_EXTENDED.

Parameters:

- **she** initialized SHE context
- **header** 16-byte cleartext counter block
- **headerSz** must be WC_SHE_KEY_SZ (16)

See:

- wc_SHE_SetKdfConstants
- wc_SHE_SetM2Header
- wc_SHE_GenerateM4M5

Return:

- 0 on success
- BAD_FUNC_ARG if she or header is NULL, or headerSz is not WC_SHE_KEY_SZ

Example

```
byte header[WC_SHE_KEY_SZ] = { ... };
int ret;
ret = wc_SHE_SetM4Header(&she, header, WC_SHE_KEY_SZ);
```

C.66.2.10 function wc_SHE_ImportM1M2M3

```
int wc_SHE_ImportM1M2M3(
    wc_SHE * she,
    const byte * m1,
```

```
    word32 m1Sz,
    const byte * m2,
    word32 m2Sz,
    const byte * m3,
    word32 m3Sz
)
```

Import externally-provided M1/M2/M3 into the SHE context. Sets the generated flag so the callback for GenerateM4M5 can read M1/M2/M3 from the context to send to hardware. Requires WOLF_CRYPTO_CB and that NO_WC_SHE_IMPORT_M123 is not defined.

Parameters:

- **she** initialized SHE context
- **m1** 16-byte M1 message (UID | KeyID | AuthID)
- **m1Sz** must be WC_SHE_M1_SZ (16)
- **m2** 32-byte M2 message (encrypted counter|flags|pad|newkey)
- **m2Sz** must be WC_SHE_M2_SZ (32)
- **m3** 16-byte M3 message (CMAC over M1|M2)
- **m3Sz** must be WC_SHE_M3_SZ (16)

See:

- wc_SHE_GenerateM1M2M3
- wc_SHE_GenerateM4M5
- wc_SHE_LoadKey

Return:

- 0 on success
- BAD_FUNC_ARG if she is NULL or any message size is incorrect

Example

```
int ret;
ret = wc_SHE_ImportM1M2M3(&she,
          m1, WC_SHE_M1_SZ,
          m2, WC_SHE_M2_SZ,
          m3, WC_SHE_M3_SZ);
```

C.66.2.11 function wc_SHE_GenerateM1M2M3

```
int wc_SHE_GenerateM1M2M3(
    wc_SHE * she,
    const byte * uid,
    word32 uidSz,
    byte authKeyId,
    const byte * authKey,
    word32 authKeySz,
    byte targetKeyId,
    const byte * newKey,
    word32 newKeySz,
    word32 counter,
    byte flags,
    byte * m1,
    word32 m1Sz,
    byte * m2,
    word32 m2Sz,
    byte * m3,
```

```
    word32 m3Sz
)
```

Generate SHE key update messages M1, M2, and M3 and write them to caller-provided buffers. Uses Miyaguchi-Preneel AES-128 KDF to derive K1 and K2 from the authorizing key, AES-CBC to encrypt the new key (M2), and AES-CMAC for authentication (M3).

Parameters:

- **she** initialized SHE context
- **uid** 15-byte SHE UID (120-bit ECU/module identifier)
- **uidSz** must be WC_SHE_UID_SZ (15)
- **authKeyId** 4-bit slot ID of the authorizing key
- **authKey** 16-byte value of the authorizing key
- **authKeySz** must be WC_SHE_KEY_SZ (16)
- **targetKeyId** 4-bit slot ID of the key being loaded
- **newKey** 16-byte value of the new key to load
- **newKeySz** must be WC_SHE_KEY_SZ (16)
- **counter** 28-bit monotonic counter value (must be strictly greater than the counter stored in the target slot)
- **flags** key protection flags (lower 4 bits)
- **m1** output buffer for M1 (16 bytes)
- **m1Sz** size of m1 buffer, must be >= WC_SHE_M1_SZ
- **m2** output buffer for M2 (32 bytes)
- **m2Sz** size of m2 buffer, must be >= WC_SHE_M2_SZ
- **m3** output buffer for M3 (16 bytes)
- **m3Sz** size of m3 buffer, must be >= WC_SHE_M3_SZ

See:

- wc_SHE_GenerateM4M5
- wc_SHE_ImportM1M2M3
- wc_SHE_LoadKey

Return:

- 0 on success
- BAD_FUNC_ARG if any required pointer is NULL, sizes are incorrect, or counter, flags, authKeyId or targetKeyId is wider than its packed field

Example

```
byte m1[WC_SHE_M1_SZ], m2[WC_SHE_M2_SZ], m3[WC_SHE_M3_SZ];
int ret;
ret = wc_SHE_GenerateM1M2M3(&she,
          uid, WC_SHE_UID_SZ,
          authKeyId, authKey, WC_SHE_KEY_SZ,
          targetKeyId, newKey, WC_SHE_KEY_SZ,
          counter, flags,
          m1, WC_SHE_M1_SZ,
          m2, WC_SHE_M2_SZ,
          m3, WC_SHE_M3_SZ);
```

C.66.2.12 function wc_SHE_GenerateM4M5

```
int wc_SHE_GenerateM4M5(
    wc_SHE * she,
    const byte * uid,
    word32 uidSz,
```

```
    byte authKeyId,
    byte targetKeyId,
    const byte * newKey,
    word32 newKeySz,
    word32 counter,
    byte * m4,
    word32 m4Sz,
    byte * m5,
    word32 m5Sz
)
```

Generate SHE verification messages M4 and M5 and write them to caller-provided buffers. Uses Miyaguchi-Preneel AES-128 KDF to derive K3 and K4 from the new key, AES-ECB for the M4 counter block, and AES-CMAC for M5. Independent of M1/M2/M3 and can be called on a separate context.

Parameters:

- **she** initialized SHE context
- **uid** 15-byte SHE UID (same UID used for M1)
- **uidSz** must be WC_SHE_UID_SZ (15)
- **authKeyId** 4-bit slot ID of the authorizing key (same as in M1)
- **targetKeyId** 4-bit slot ID of the key being loaded (same as in M1)
- **newKey** 16-byte value of the new key
- **newKeySz** must be WC_SHE_KEY_SZ (16)
- **counter** 28-bit monotonic counter (same value as in M2)
- **m4** output buffer for M4 (32 bytes)
- **m4Sz** size of m4 buffer, must be >= WC_SHE_M4_SZ
- **m5** output buffer for M5 (16 bytes)
- **m5Sz** size of m5 buffer, must be >= WC_SHE_M5_SZ

See:

- wc_SHE_GenerateM1M2M3
- wc_SHE_LoadKey_Verify

Return:

- 0 on success
- BAD_FUNC_ARG if any required pointer is NULL, sizes are incorrect, or counter, authKeyId or targetKeyId is wider than its packed field

Example

```
byte m4[WC_SHE_M4_SZ], m5[WC_SHE_M5_SZ];
int ret;
ret = wc_SHE_GenerateM4M5(&she,
          uid, WC_SHE_UID_SZ,
          authKeyId, targetKeyId,
          newKey, WC_SHE_KEY_SZ,
          counter,
          m4, WC_SHE_M4_SZ,
          m5, WC_SHE_M5_SZ);
```

C.66.2.13 function wc_SHE_LoadKey

```
int wc_SHE_LoadKey(
    void * heap,
    int devId,
    const byte * m1,
```

```
    word32 m1Sz,
    const byte * m2,
    word32 m2Sz,
    const byte * m3,
    word32 m3Sz,
    byte * m4,
    word32 m4Sz,
    byte * m5,
    word32 m5Sz
)
```

One-shot convenience wrapper: Init, ImportM1M2M3, GenerateM4M5 (via callback), Free. Dispatches to a hardware crypto callback that sends M1/M2/M3 to the HSM and returns M4/M5. Requires a valid devId (not INVALID_DEVID). Define NO_WC_SHE_LOADKEY to compile out.

Parameters:

- **heap** heap hint for internal allocations, or NULL
- **devId** crypto callback device ID (must not be INVALID_DEVID)
- **m1** 16-byte M1 input message
- **m1Sz** must be WC_SHE_M1_SZ (16)
- **m2** 32-byte M2 input message
- **m2Sz** must be WC_SHE_M2_SZ (32)
- **m3** 16-byte M3 input message
- **m3Sz** must be WC_SHE_M3_SZ (16)
- **m4** output buffer for M4 (32 bytes)
- **m4Sz** size of m4 buffer, must be >= WC_SHE_M4_SZ
- **m5** output buffer for M5 (16 bytes)
- **m5Sz** size of m5 buffer, must be >= WC_SHE_M5_SZ

See:

- wc_SHE_LoadKey_Id
- wc_SHE_LoadKey_Label
- wc_SHE_LoadKey_Verify
- wc_SHE_ImportM1M2M3
- wc_SHE_GenerateM4M5

Return:

- 0 on success
- BAD_FUNC_ARG if any required pointer is NULL or sizes are incorrect

Example

```
byte m4[WC_SHE_M4_SZ], m5[WC_SHE_M5_SZ];
int ret;
ret = wc_SHE_LoadKey(NULL, myDevId,
          m1, WC_SHE_M1_SZ,
          m2, WC_SHE_M2_SZ,
          m3, WC_SHE_M3_SZ,
          m4, WC_SHE_M4_SZ,
          m5, WC_SHE_M5_SZ);
```

C.66.2.14 function wc_SHE_LoadKey_Id

```
int wc_SHE_LoadKey_Id(
    unsigned char * id,
    int idLen,
```

```
    void * heap,
    int devId,
    const byte * m1,
    word32 m1Sz,
    const byte * m2,
    word32 m2Sz,
    const byte * m3,
    word32 m3Sz,
    byte * m4,
    word32 m4Sz,
    byte * m5,
    word32 m5Sz
)
```

One-shot Load Key with an opaque hardware key identifier. Same as wc_SHE_LoadKey but initializes the context with wc_SHE_Init_Id. Define NO_WC_SHE_LOADKEY to compile out.

Parameters:

- **id** opaque key identifier bytes
- **idLen** length of id in bytes
- **heap** heap hint for internal allocations, or NULL
- **devId** crypto callback device ID (must not be INVALID_DEVID)
- **m1** 16-byte M1 input message
- **m1Sz** must be WC_SHE_M1_SZ (16)
- **m2** 32-byte M2 input message
- **m2Sz** must be WC_SHE_M2_SZ (32)
- **m3** 16-byte M3 input message
- **m3Sz** must be WC_SHE_M3_SZ (16)
- **m4** output buffer for M4 (32 bytes)
- **m4Sz** size of m4 buffer, must be >= WC_SHE_M4_SZ
- **m5** output buffer for M5 (16 bytes)
- **m5Sz** size of m5 buffer, must be >= WC_SHE_M5_SZ

See:

- wc_SHE_LoadKey
- wc_SHE_LoadKey_Label
- wc_SHE_LoadKey_Verify_Id

Return:

- 0 on success
- BAD_FUNC_ARG if any required pointer is NULL or sizes are incorrect

Example

```
byte m4[WC_SHE_M4_SZ], m5[WC_SHE_M5_SZ];
unsigned char keyId[] = { 0x01, 0x02 };
int ret;
ret = wc_SHE_LoadKey_Id(keyId, sizeof(keyId), NULL, myDevId,
          m1, WC_SHE_M1_SZ,
          m2, WC_SHE_M2_SZ,
          m3, WC_SHE_M3_SZ,
          m4, WC_SHE_M4_SZ,
          m5, WC_SHE_M5_SZ);
```

C.66.2.15 function wc_SHE_LoadKey_Label

```
int wc_SHE_LoadKey_Label(
    const char * label,
    void * heap,
    int devId,
    const byte * m1,
    word32 m1Sz,
    const byte * m2,
    word32 m2Sz,
    const byte * m3,
    word32 m3Sz,
    byte * m4,
    word32 m4Sz,
    byte * m5,
    word32 m5Sz
)
```

One-shot Load Key with a human-readable key label. Same as wc_SHE_LoadKey but initializes the context with wc_SHE_Init_Label. Define NO_WC_SHE_LOADKEY to compile out.

Parameters:

- **label** NUL-terminated key label string
- **heap** heap hint for internal allocations, or NULL
- **devId** crypto callback device ID (must not be INVALID_DEVID)
- **m1** 16-byte M1 input message
- **m1Sz** must be WC_SHE_M1_SZ (16)
- **m2** 32-byte M2 input message
- **m2Sz** must be WC_SHE_M2_SZ (32)
- **m3** 16-byte M3 input message
- **m3Sz** must be WC_SHE_M3_SZ (16)
- **m4** output buffer for M4 (32 bytes)
- **m4Sz** size of m4 buffer, must be >= WC_SHE_M4_SZ
- **m5** output buffer for M5 (16 bytes)
- **m5Sz** size of m5 buffer, must be >= WC_SHE_M5_SZ

See:

- wc_SHE_LoadKey
- wc_SHE_LoadKey_Id
- wc_SHE_LoadKey_Verify_Label

Return:

- 0 on success
- BAD_FUNC_ARG if any required pointer is NULL or sizes are incorrect

Example

```
byte m4[WC_SHE_M4_SZ], m5[WC_SHE_M5_SZ];
int ret;
ret = wc_SHE_LoadKey_Label("ecu-master", NULL, myDevId,
          m1, WC_SHE_M1_SZ,
          m2, WC_SHE_M2_SZ,
          m3, WC_SHE_M3_SZ,
          m4, WC_SHE_M4_SZ,
          m5, WC_SHE_M5_SZ);
```

C.66.2.16 function wc_SHE_LoadKey_Verify

```
int wc_SHE_LoadKey_Verify(
    void * heap,
    int devId,
    const byte * m1,
    word32 m1Sz,
    const byte * m2,
    word32 m2Sz,
    const byte * m3,
    word32 m3Sz,
    byte * m4,
    word32 m4Sz,
    byte * m5,
    word32 m5Sz,
    const byte * m4Expected,
    word32 m4ExpectedSz,
    const byte * m5Expected,
    word32 m5ExpectedSz
)
```

One-shot Load Key with M4/M5 verification. Same as wc_SHE_LoadKey but also compares the M4/M5 returned by the HSM against caller-provided expected values using constant-time comparison. Returns SIG_VERIFY_E on mismatch. The actual M4/M5 are still written to the output buffers on failure. Define NO_WC_SHE_LOADKEY to compile out.

Parameters:

- **heap** heap hint for internal allocations, or NULL
- **devId** crypto callback device ID (must not be INVALID_DEVID)
- **m1** 16-byte M1 input message
- **m1Sz** must be WC_SHE_M1_SZ (16)
- **m2** 32-byte M2 input message
- **m2Sz** must be WC_SHE_M2_SZ (32)
- **m3** 16-byte M3 input message
- **m3Sz** must be WC_SHE_M3_SZ (16)
- **m4** output buffer for M4 (32 bytes)
- **m4Sz** size of m4 buffer, must be >= WC_SHE_M4_SZ
- **m5** output buffer for M5 (16 bytes)
- **m5Sz** size of m5 buffer, must be >= WC_SHE_M5_SZ
- **m4Expected** expected M4 verification message to compare against
- **m4ExpectedSz** must be WC_SHE_M4_SZ (32)
- **m5Expected** expected M5 verification message to compare against
- **m5ExpectedSz** must be WC_SHE_M5_SZ (16)

See:

- wc_SHE_LoadKey
- wc_SHE_LoadKey_Verify_Id
- wc_SHE_LoadKey_Verify_Label

Return:

- 0 on success
- SIG_VERIFY_E if M4/M5 do not match expected values
- BAD_FUNC_ARG if any required pointer is NULL or sizes are incorrect

Example

```
byte m4[WC_SHE_M4_SZ], m5[WC_SHE_M5_SZ];
int ret;
```

```
ret = wc_SHE_LoadKey_Verify(NULL, myDevId,
          m1, WC_SHE_M1_SZ,
          m2, WC_SHE_M2_SZ,
          m3, WC_SHE_M3_SZ,
          m4, WC_SHE_M4_SZ,
          m5, WC_SHE_M5_SZ,
          expectedM4, WC_SHE_M4_SZ,
          expectedM5, WC_SHE_M5_SZ);
if (ret == SIG_VERIFY_E) {
    // M4/M5 mismatch
}
```

C.66.2.17 function wc_SHE_LoadKey_Verify_Id

```
int wc_SHE_LoadKey_Verify_Id(
    unsigned char * id,
    int idLen,
    void * heap,
    int devId,
    const byte * m1,
    word32 m1Sz,
    const byte * m2,
    word32 m2Sz,
    const byte * m3,
    word32 m3Sz,
    byte * m4,
    word32 m4Sz,
    byte * m5,
    word32 m5Sz,
    const byte * m4Expected,
    word32 m4ExpectedSz,
    const byte * m5Expected,
    word32 m5ExpectedSz
)
```

One-shot Load Key with opaque key identifier and M4/M5 verification. Combines wc_SHE_LoadKey_Id and wc_SHE_LoadKey_Verify. Define NO_WC_SHE_LOADKEY to compile out.

Parameters:

- **id** opaque key identifier bytes
- **idLen** length of id in bytes
- **heap** heap hint for internal allocations, or NULL
- **devId** crypto callback device ID (must not be INVALID_DEVID)
- **m1** 16-byte M1 input message
- **m1Sz** must be WC_SHE_M1_SZ (16)
- **m2** 32-byte M2 input message
- **m2Sz** must be WC_SHE_M2_SZ (32)
- **m3** 16-byte M3 input message
- **m3Sz** must be WC_SHE_M3_SZ (16)
- **m4** output buffer for M4 (32 bytes)
- **m4Sz** size of m4 buffer, must be >= WC_SHE_M4_SZ
- **m5** output buffer for M5 (16 bytes)
- **m5Sz** size of m5 buffer, must be >= WC_SHE_M5_SZ
- **m4Expected** expected M4 verification message
- **m4ExpectedSz** must be WC_SHE_M4_SZ (32)

- **m5Expected** expected M5 verification message
- **m5ExpectedSz** must be WC_SHE_M5_SZ (16)

See:

- wc_SHE_LoadKey_Id
- wc_SHE_LoadKey_Verify
- wc_SHE_LoadKey_Verify_Label

Return:

- 0 on success
- SIG_VERIFY_E if M4/M5 do not match expected values
- BAD_FUNC_ARG if any required pointer is NULL or sizes are incorrect

Example

```
byte m4[WC_SHE_M4_SZ], m5[WC_SHE_M5_SZ];
unsigned char keyId[] = { 0x01, 0x02 };
int ret;
ret = wc_SHE_LoadKey_Verify_Id(keyId, sizeof(keyId), NULL, myDevId,
          m1, WC_SHE_M1_SZ, m2, WC_SHE_M2_SZ, m3, WC_SHE_M3_SZ,
          m4, WC_SHE_M4_SZ, m5, WC_SHE_M5_SZ,
          expectedM4, WC_SHE_M4_SZ, expectedM5, WC_SHE_M5_SZ);
```

C.66.2.18 function wc_SHE_LoadKey_Verify_Label

```
int wc_SHE_LoadKey_Verify_Label(
    const char * label,
    void * heap,
    int devId,
    const byte * m1,
    word32 m1Sz,
    const byte * m2,
    word32 m2Sz,
    const byte * m3,
    word32 m3Sz,
    byte * m4,
    word32 m4Sz,
    byte * m5,
    word32 m5Sz,
    const byte * m4Expected,
    word32 m4ExpectedSz,
    const byte * m5Expected,
    word32 m5ExpectedSz
)
```

One-shot Load Key with key label and M4/M5 verification. Combines wc_SHE_LoadKey_Label and wc_SHE_LoadKey_Verify. Define NO_WC_SHE_LOADKEY to compile out.

Parameters:

- **label** NUL-terminated key label string
- **heap** heap hint for internal allocations, or NULL
- **devId** crypto callback device ID (must not be INVALID_DEVID)
- **m1** 16-byte M1 input message
- **m1Sz** must be WC_SHE_M1_SZ (16)
- **m2** 32-byte M2 input message
- **m2Sz** must be WC_SHE_M2_SZ (32)

- **m3** 16-byte M3 input message
- **m3Sz** must be WC_SHE_M3_SZ (16)
- **m4** output buffer for M4 (32 bytes)
- **m4Sz** size of m4 buffer, must be >= WC_SHE_M4_SZ
- **m5** output buffer for M5 (16 bytes)
- **m5Sz** size of m5 buffer, must be >= WC_SHE_M5_SZ
- **m4Expected** expected M4 verification message
- **m4ExpectedSz** must be WC_SHE_M4_SZ (32)
- **m5Expected** expected M5 verification message
- **m5ExpectedSz** must be WC_SHE_M5_SZ (16)

See:

- wc_SHE_LoadKey_Label
- wc_SHE_LoadKey_Verify
- wc_SHE_LoadKey_Verify_Id

Return:

- 0 on success
- SIG_VERIFY_E if M4/M5 do not match expected values
- BAD_FUNC_ARG if any required pointer is NULL or sizes are incorrect

Example

```
byte m4[WC_SHE_M4_SZ], m5[WC_SHE_M5_SZ];
int ret;
ret = wc_SHE_LoadKey_Verify_Label("ecu-master", NULL, myDevId,
          m1, WC_SHE_M1_SZ, m2, WC_SHE_M2_SZ, m3, WC_SHE_M3_SZ,
          m4, WC_SHE_M4_SZ, m5, WC_SHE_M5_SZ,
          expectedM4, WC_SHE_M4_SZ, expectedM5, WC_SHE_M5_SZ);
```

C.66.2.19 function wc_SHE_ExportKey

```
int wc_SHE_ExportKey(
    wc_SHE * she,
    byte * m1,
    word32 m1Sz,
    byte * m2,
    word32 m2Sz,
    byte * m3,
    word32 m3Sz,
    byte * m4,
    word32 m4Sz,
    byte * m5,
    word32 m5Sz,
    const void * ctx
)
```

Export a key from hardware in SHE loadable format (M1-M5). Some HSMs allow exporting certain key slots (e.g. RAM key) so they can be re-loaded later via the SHE key update protocol. Requires WOLF_CRYPTO_CB and that NO_WC_SHE_EXPORTKEY is not defined. Any output buffer may be NULL to skip that message.

Parameters:

- **she** initialized SHE context
- **m1** output buffer for M1 (16 bytes), or NULL to skip
- **m1Sz** size of m1 buffer

- **m2** output buffer for M2 (32 bytes), or NULL to skip
- **m2Sz** size of m2 buffer
- **m3** output buffer for M3 (16 bytes), or NULL to skip
- **m3Sz** size of m3 buffer
- **m4** output buffer for M4 (32 bytes), or NULL to skip
- **m4Sz** size of m4 buffer
- **m5** output buffer for M5 (16 bytes), or NULL to skip
- **m5Sz** size of m5 buffer
- **ctx** read-only caller context passed to the callback

See:

- wc_SHE_ImportM1M2M3
- wc_SHE_LoadKey

Return:

- 0 on success
- BAD_FUNC_ARG if she is NULL
- CRYPTOCB_UNAVAILABLE if no callback is registered

Example

```
byte m1[WC_SHE_M1_SZ], m2[WC_SHE_M2_SZ], m3[WC_SHE_M3_SZ];
byte m4[WC_SHE_M4_SZ], m5[WC_SHE_M5_SZ];
int ret;
ret = wc_SHE_ExportKey(&she,
          m1, WC_SHE_M1_SZ,
          m2, WC_SHE_M2_SZ,
          m3, WC_SHE_M3_SZ,
          m4, WC_SHE_M4_SZ,
          m5, WC_SHE_M5_SZ,
          NULL);
```

C.66.2.20 function wc_SHE_AesMp16

```
int wc_SHE_AesMp16(
    Aes * aes,
    const byte * in,
    word32 inSz,
    byte * out
)
```

Miyaguchi-Preneel AES-128 one-way compression function. H_0 = 0, H_i = E_{H_{i-1}}(M_i) XOR M_i XOR H_{i-1}. Only valid for AES-128 where key size equals block size. This is an internal function exposed for testing purposes.

Parameters:

- **aes** caller-owned, already-initialized Aes structure
- **in** input data (e.g. BaseKey || KDF_Constant, 32 bytes)
- **inSz** length of input in bytes (zero-padded to block boundary)
- **out** output buffer for 16-byte compressed result

See:

- wc_SHE_GenerateM1M2M3
- wc_SHE_GenerateM4M5

Return:

- 0 on success
- BAD_FUNC_ARG if any pointer is NULL

Example

```
Aes aes;
byte input[32] = { ... };
byte output[WC_SHE_KEY_SZ];
int ret;
wc_AesInit(&aes, NULL, INVALID_DEVID);
ret = wc_SHE_AesMp16(&aes, input, sizeof(input), output);
wc_AesFree(&aes);
```

C.66.3 Source code

```
int wc_SHE_Init(wc_SHE* she, void* heap, int devId);

int wc_SHE_Init_Id(wc_SHE* she, unsigned char* id, int len,
                   void* heap, int devId);

int wc_SHE_Init_Label(wc_SHE* she, const char* label,
                      void* heap, int devId);

void wc_SHE_Free(wc_SHE* she);

int wc_SHE_GetUID(wc_SHE* she, byte* uid, word32 uidSz,
                  const void* ctx);

int wc_SHE_GetCounter(wc_SHE* she, word32* counter,
                      const void* ctx);

int wc_SHE_SetKdfConstants(wc_SHE* she,
                           const byte* encC, word32 encCSz,
                           const byte* macC, word32 macCSz);

int wc_SHE_SetM2Header(wc_SHE* she,
                       const byte* header, word32 headerSz);

int wc_SHE_SetM4Header(wc_SHE* she,
                       const byte* header, word32 headerSz);

int wc_SHE_ImportM1M2M3(wc_SHE* she,
                        const byte* m1, word32 m1Sz,
                        const byte* m2, word32 m2Sz,
                        const byte* m3, word32 m3Sz);

int wc_SHE_GenerateM1M2M3(wc_SHE* she,
                      const byte* uid, word32 uidSz,
                      byte authKeyId, const byte* authKey, word32 authKeySz,
                      byte targetKeyId, const byte* newKey, word32 newKeySz,
                      word32 counter, byte flags,
                      byte* m1, word32 m1Sz,
                      byte* m2, word32 m2Sz,
                      byte* m3, word32 m3Sz);
```

```
int wc_SHE_GenerateM4M5(wc_SHE* she,
                        const byte* uid, word32 uidSz,
                        byte authKeyId, byte targetKeyId,
                        const byte* newKey, word32 newKeySz,
                        word32 counter,
                        byte* m4, word32 m4Sz,
                        byte* m5, word32 m5Sz);

int wc_SHE_LoadKey(
    void* heap, int devId,
    const byte* m1, word32 m1Sz,
    const byte* m2, word32 m2Sz,
    const byte* m3, word32 m3Sz,
    byte* m4, word32 m4Sz,
    byte* m5, word32 m5Sz);

int wc_SHE_LoadKey_Id(
    unsigned char* id, int idLen,
    void* heap, int devId,
    const byte* m1, word32 m1Sz,
    const byte* m2, word32 m2Sz,
    const byte* m3, word32 m3Sz,
    byte* m4, word32 m4Sz,
    byte* m5, word32 m5Sz);

int wc_SHE_LoadKey_Label(
    const char* label,
    void* heap, int devId,
    const byte* m1, word32 m1Sz,
    const byte* m2, word32 m2Sz,
    const byte* m3, word32 m3Sz,
    byte* m4, word32 m4Sz,
    byte* m5, word32 m5Sz);

int wc_SHE_LoadKey_Verify(
    void* heap, int devId,
    const byte* m1, word32 m1Sz,
    const byte* m2, word32 m2Sz,
    const byte* m3, word32 m3Sz,
    byte* m4, word32 m4Sz,
    byte* m5, word32 m5Sz,
    const byte* m4Expected, word32 m4ExpectedSz,
    const byte* m5Expected, word32 m5ExpectedSz);

int wc_SHE_LoadKey_Verify_Id(
    unsigned char* id, int idLen,
    void* heap, int devId,
    const byte* m1, word32 m1Sz,
    const byte* m2, word32 m2Sz,
    const byte* m3, word32 m3Sz,
    byte* m4, word32 m4Sz,
    byte* m5, word32 m5Sz,
    const byte* m4Expected, word32 m4ExpectedSz,
```

```
    const byte* m5Expected, word32 m5ExpectedSz);

int wc_SHE_LoadKey_Verify_Label(
    const char* label,
    void* heap, int devId,
    const byte* m1, word32 m1Sz,
    const byte* m2, word32 m2Sz,
    const byte* m3, word32 m3Sz,
    byte* m4, word32 m4Sz,
    byte* m5, word32 m5Sz,
    const byte* m4Expected, word32 m4ExpectedSz,
    const byte* m5Expected, word32 m5ExpectedSz);

int wc_SHE_ExportKey(wc_SHE* she,
                     byte* m1, word32 m1Sz,
                     byte* m2, word32 m2Sz,
                     byte* m3, word32 m3Sz,
                     byte* m4, word32 m4Sz,
                     byte* m5, word32 m5Sz,
                     const void* ctx);

int wc_SHE_AesMp16(Aes* aes, const byte* in, word32 inSz,
                   byte* out);
```

C.67 wc_slhdsa.h

C.67.1 Functions

	Name
int	**wc_SlhDsaKey_Init**(SlhDsaKey * key, enum SlhDsaParam param, void * heap, int devId)Initializes an SLH_DSA key object with the specified parameter set. Must be called before any other SLH_DSA operation. Use wc_SlhDsaKey_Free() to release resources when done.
int	**wc_SlhDsaKey_Init_id**(SlhDsaKey * key, enum SlhDsaParam param, const unsigned char * id, int len, void * heap, int devId)Initializes an SLH_DSA key with a device_side key identifier (id). Equivalent to wc_SlhDsaKey_Init() but also stashes a binary id that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.

	Name
int	**wc_SlhDsaKey_Init_label**(SlhDsaKey * key, enum SlhDsaParam param, const char * label, void * heap, int devId)Initializes an SLH_DSA key with a device_side key label. Equivalent to wc_SlhDsaKey_Init() but also stashes a label string that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.
void	**wc_SlhDsaKey_Free**(SlhDsaKey * key)Frees resources associated with an SLH-DSA key object.
int	**wc_SlhDsaKey_MakeKey**(SlhDsaKey * key, WC_RNG * rng)Generates a new SLH_DSA key pair using the RNG for randomness. The key must have been initialized with wc_SlhDsaKey_Init() first.
int	**wc_SlhDsaKey_MakeKeyWithRandom**(SlhDsaKey * key, const byte * sk_seed, word32 sk_seed_len, const byte * sk_prf, word32 sk_prf_len, const byte * pk_seed, word32 pk_seed_len)Generates an SLH-DSA key pair from caller-provided seed material. This is the deterministic key generation interface — given the same seeds, the same key pair is produced.
int	**wc_SlhDsaKey_SignDeterministic**(SlhDsaKey * key, const byte * ctx, byte ctxSz, const byte * msg, word32 msgSz, byte * sig, word32 * sigSz)Signs a message using the SLH_DSA external (pure) interface with deterministic randomness. This is FIPS 205 Algorithm 22 with opt_rand set to PK.seed. The message M is wrapped internally as M' = 0x00
int	**wc_SlhDsaKey_SignWithRandom**(SlhDsaKey * key, const byte * ctx, byte ctxSz, const byte * msg, word32 msgSz, byte * sig, word32 * sigSz, const byte * addRnd)Signs a message using the SLH_DSA external (pure) interface with caller-provided additional randomness. This is FIPS 205 Algorithm 22 with an explicit opt_rand value.
int	**wc_SlhDsaKey_Sign**(SlhDsaKey * key, const byte * ctx, byte ctxSz, const byte * msg, word32 msgSz, byte * sig, word32 * sigSz, WC_RNG * rng)Signs a message using the SLH_DSA external (pure) interface with RNG-provided randomness. This is the general-purpose signing function that uses the WC_RNG for opt_rand.

	Name
int	**wc_SlhDsaKey_Verify**(SlhDsaKey * key, const byte * ctx, byte ctxSz, const byte * msg, word32 msgSz, const byte * sig, word32 sigSz)Verifies an SLH_DSA signature over a message using the external (pure) interface. This is FIPS 205 Algorithm 24. The message is wrapped internally as M' = 0x00
int	**wc_SlhDsaKey_SignMsgDeterministic**(SlhDsaKey * key, const byte * mprime, word32 mprimeSz, byte * sig, word32 * sigSz)Signs using the SLH_DSA internal interface with deterministic randomness. Unlike the external interface, M' is provided directly by the caller — no wrapping is performed. This corresponds to FIPS 205 Algorithm 19 (slh_sign_internal) with opt_rand set to PK.seed.
int	**wc_SlhDsaKey_SignMsgWithRandom**(SlhDsaKey * key, const byte * mprime, word32 mprimeSz, byte * sig, word32 * sigSz, const byte * addRnd)Signs using the SLH_DSA internal interface with caller_provided additional randomness. M' is provided directly — no wrapping is performed. This corresponds to FIPS 205 Algorithm 19 (slh_sign_internal) with an explicit opt_rand value. See wc_SlhDsaKey_SignMsgDeterministic for the M' layout used by HashSLH-DSA.
int	**wc_SlhDsaKey_VerifyMsg**(SlhDsaKey * key, const byte * mprime, word32 mprimeSz, const byte * sig, word32 sigSz)Verifies an SLH_DSA signature using the internal interface. M' is provided directly — no wrapping is performed. This corresponds to FIPS 205 Algorithm 20 (slh_verify_internal).
int	**wc_SlhDsaKey_SignHashDeterministic**(SlhDsaKey * key, const byte * ctx, byte ctxSz, const byte * hash, word32 hashSz, enum wc_HashType hashType, byte * sig, word32 * sigSz)Signs a caller_pre_hashed message digest using the SLH_DSA external (HashSLH_DSA) interface with deterministic randomness, per FIPS 205 Algorithm 23 with the pre_hash domain separator (0x01). The caller must hash the application message with hashType first and pass the digest as hash; this function does NOT hash its input.

	Name
int	**wc_SlhDsaKey_SignHashWithRandom**(SlhDsaKey * key, const byte * ctx, byte ctxSz, const byte * hash, word32 hashSz, enum wc_HashType hashType, byte * sig, word32 * sigSz, const byte * addRnd)Signs a caller_pre_hashed message digest using the SLH_DSA external (HashSLH_DSA) interface with caller-provided additional randomness. The caller must hash the application message with hashType first and pass the digest as hash; this function does NOT hash its input.
int	**wc_SlhDsaKey_SignHash**(SlhDsaKey * key, const byte * ctx, byte ctxSz, const byte * hash, word32 hashSz, enum wc_HashType hashType, byte * sig, word32 * sigSz, WC_RNG * rng)Signs a caller_pre_hashed message digest using the SLH_DSA external (HashSLH_DSA) interface with RNG-provided randomness. The caller must hash the application message with hashType first and pass the digest as hash; this function does NOT hash its input.
int	**wc_SlhDsaKey_VerifyHash**(SlhDsaKey * key, const byte * ctx, byte ctxSz, const byte * hash, word32 hashSz, enum wc_HashType hashType, const byte * sig, word32 sigSz)Verifies an SLH_DSA signature using the external HashSLH_DSA interface (FIPS 205 Algorithm 25). The caller must hash the application message with hashType first and pass the digest as hash; this function does NOT hash its input.
int	**wc_SlhDsaKey_ImportPrivate**(SlhDsaKey * key, const byte * in, word32 inLen)Imports an SLH_DSA private key from a raw byte buffer. The buffer must contain the full private key (4*n bytes: SK.seed
int	**wc_SlhDsaKey_ImportPublic**(SlhDsaKey * key, const byte * in, word32 inLen)Imports an SLH_DSA public key from a raw byte buffer. The buffer must contain PK.seed
int	**wc_SlhDsaKey_CheckKey**(SlhDsaKey * key)Checks the consistency of an SLH-DSA key. For a key with both private and public components, verifies that the public key matches the private key.
int	**wc_SlhDsaKey_ExportPrivate**(SlhDsaKey * key, byte * out, word32 * outLen)Exports the private key from an SLH_DSA key object into a raw byte buffer (4*n bytes).
int	**wc_SlhDsaKey_ExportPublic**(SlhDsaKey * key, byte * out, word32 * outLen)Exports the public key from an SLH_DSA key object into a raw byte buffer (2*n bytes: PK.seed

	Name
int	**wc_SlhDsaKey_PrivateSize**(SlhDsaKey * key)Returns the private key size in bytes for the key's parameter set.
int	**wc_SlhDsaKey_PublicSize**(SlhDsaKey * key)Returns the public key size in bytes for the key's parameter set.
int	**wc_SlhDsaKey_SigSize**(SlhDsaKey * key)Returns the signature size in bytes for the key's parameter set.
int	**wc_SlhDsaKey_PrivateSizeFromParam**(enum SlhDsaParam param)Returns the private key size in bytes for the given parameter set without needing an initialized key object.
int	**wc_SlhDsaKey_PublicSizeFromParam**(enum SlhDsaParam param)Returns the public key size in bytes for the given parameter set without needing an initialized key object.
int	**wc_SlhDsaKey_SigSizeFromParam**(enum SlhDsaParam param)Returns the signature size in bytes for the given parameter set without needing an initialized key object.
int	**wc_SlhDsaKey_PrivateKeyDecode**(const byte * input, word32 * inOutIdx, SlhDsaKey * key, word32 inSz)Decodes a DER_encoded SLH_DSA private key in the PKCS#8 OneAsymmetricKey format defined by RFC 9909. The privateKey OCTET STRING contains the raw concatenation SK.seed
int	**wc_SlhDsaKey_PublicKeyDecode**(const byte * input, word32 * inOutIdx, SlhDsaKey * key, word32 inSz)Decodes a DER_encoded SLH_DSA public key in the SubjectPublicKeyInfo (SPKI) format. The SLH-DSA parameter set is detected from the AlgorithmIdentifier OID and key->params is updated accordingly.
int	**wc_SlhDsaKey_KeyToDer**(SlhDsaKey * key, byte * output, word32 inLen)Encodes an SLH_DSA private key to DER in the PKCS#8 OneAsymmetricKey format defined by RFC 9909. The privateKey OCTET STRING contains the raw 4*n bytes (SK.seed
int	**wc_SlhDsaKey_PrivateKeyToDer**(SlhDsaKey * key, byte * output, word32 inLen)Encodes an SLH-DSA private key to DER. RFC 9909 packs SK.seed

	Name
int	**wc_SlhDsaKey_PublicKeyToDer**(SlhDsaKey * key, byte * output, word32 inLen, int withAlg)Encodes an SLH_DSA public key to DER. When withAlg is non_zero the output is a full SubjectPublicKeyInfo structure (AlgorithmIdentifier plus BIT STRING). When withAlg is zero the output contains the raw public key bytes without the SPKI wrapping.

C.67.2 Functions Documentation

C.67.2.1 function wc_SlhDsaKey_Init

```
int wc_SlhDsaKey_Init(
    SlhDsaKey * key,
    enum SlhDsaParam param,
    void * heap,
    int devId
)
```

Initializes an SLH-DSA key object with the specified parameter set. Must be called before any other SLH-DSA operation. Use wc_SlhDsaKey_Free() to release resources when done.

Parameters:

- **key** Pointer to the SlhDsaKey to initialize.
- **param** Parameter set to use. One of: SLHDSA_SHAKE128S, SLHDSA_SHAKE128F, SLHDSA_SHAKE192S, SLHDSA_SHAKE192F, SLHDSA_SHAKE256S, SLHDSA_SHAKE256F, SLHDSA_SHA2_128S, SLHDSA_SHA2_128F, SLHDSA_SHA2_192S, SLHDSA_SHA2_192F, SLHDSA_SHA2_256S, SLHDSA_SHA2_256F.
- **heap** Pointer to heap hint for dynamic memory allocation. May be NULL.
- **devId** Device identifier for hardware crypto callbacks. Use INVALID_DEVID for software-only.

See:

- wc_SlhDsaKey_Free
- wc_SlhDsaKey_MakeKey

Return:

- 0 on success.
- BAD_FUNC_ARG if key is NULL or param is invalid.

SLH-DSA (FIPS 205) is a stateless hash-based digital signature algorithm. Parameter sets control the hash function (SHAKE or SHA2), security level (128, 192, 256), and speed/size tradeoff (s = small signatures, f = fast signing).

Example

```
SlhDsaKey key;
int ret;

ret = wc_SlhDsaKey_Init(&key, SLHDSA_SHAKE128F, NULL, INVALID_DEVID);
if (ret != 0) {
    // error initializing key
}
```

```
// ... use key ...
wc_SlhDsaKey_Free(&key);
```

C.67.2.2 function wc_SlhDsaKey_Init_id

```
int wc_SlhDsaKey_Init_id(
    SlhDsaKey * key,
    enum SlhDsaParam param,
    const unsigned char * id,
    int len,
    void * heap,
    int devId
)
```

Initializes an SLH-DSA key with a device-side key identifier (id). Equivalent to wc_SlhDsaKey_Init() but also stashes a binary id that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.

Parameters:

- **key** Pointer to the SlhDsaKey to initialize.
- **param** Parameter set to use (see wc_SlhDsaKey_Init).
- **id** Pointer to the device-side key identifier bytes. May be NULL if len is 0.
- **len** Number of bytes in id. Must be in [0, SLHDSA_MAX_ID_LEN].
- **heap** Pointer to heap hint for dynamic memory allocation. May be NULL.
- **devId** Device identifier for the crypto callback. Should be a registered cb devId, not INVALID_DEVID, for the id to be meaningful.

See:

- wc_SlhDsaKey_Init
- wc_SlhDsaKey_Init_label
- wc_SlhDsaKey_Free

Return:

- 0 on success.
- BAD_FUNC_ARG if key is NULL, or if id is NULL while len > 0.
- BUFFER_E if len is negative or greater than SLHDSA_MAX_ID_LEN.

The id is copied into the key object; the caller may free its buffer immediately after this call returns.

Example

```
SlhDsaKey key;
unsigned char id[8] = { 0x01, 0x02, 0x03, 0x04,
                        0x05, 0x06, 0x07, 0x08 };
int ret;

ret = wc_SlhDsaKey_Init_id(&key, SLHDSA_SHAKE128F, id, sizeof(id),
    NULL, devId);
if (ret != 0) {
    // error initializing key with id
}
// ... use key, the cb resolves id -> device key ...
wc_SlhDsaKey_Free(&key);
```

C.67.2.3 function wc_SlhDsaKey_Init_label

```
int wc_SlhDsaKey_Init_label(
    SlhDsaKey * key,
    enum SlhDsaParam param,
    const char * label,
    void * heap,
    int devId
)
```

Initializes an SLH-DSA key with a device-side key label. Equivalent to wc_SlhDsaKey_Init() but also stashes a label string that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.

Parameters:

- **key** Pointer to the SlhDsaKey to initialize.
- **param** Parameter set to use (see wc_SlhDsaKey_Init).
- **label** NUL-terminated device-side key label string.
- **heap** Pointer to heap hint for dynamic memory allocation. May be NULL.
- **devId** Device identifier for the crypto callback. Should be a registered cb devId, not INVALID_DEVID, for the label to be meaningful.

See:

- wc_SlhDsaKey_Init
- wc_SlhDsaKey_Init_id
- wc_SlhDsaKey_Free

Return:

- 0 on success.
- BAD_FUNC_ARG if key or label is NULL.
- BUFFER_E if label is empty or longer than SLHDSA_MAX_LABEL_LEN.

The label length is taken via XSTRLEN, so embedded NUL bytes terminate the label. The copy stored in key->label is NOT guaranteed to be NUL-terminated (when the input is exactly SLHDSA_MAX_LABEL_LEN bytes the entire array is consumed). Consumers must read at most key->labelLen bytes — do not pass key->label to C-string APIs.

Example

```
SlhDsaKey key;
int ret;

ret = wc_SlhDsaKey_Init_label(&key, SLHDSA_SHAKE128F,
    "device-key-1", NULL, devId);
if (ret != 0) {
    // error initializing key with label
}
// ... use key, the cb resolves label -> device key ...
wc_SlhDsaKey_Free(&key);
```

C.67.2.4 function wc_SlhDsaKey_Free

```
void wc_SlhDsaKey_Free(
    SlhDsaKey * key
)
```

Frees resources associated with an SLH-DSA key object.

Parameters:

- **key** Pointer to the SlhDsaKey to free. May be NULL.

See: wc_SlhDsaKey_Init

Example

```
SlhDsaKey key;
wc_SlhDsaKey_Init(&key, SLHDSA_SHAKE128F, NULL, INVALID_DEVID);
// ... use key ...
wc_SlhDsaKey_Free(&key);
```

C.67.2.5 function wc_SlhDsaKey_MakeKey

```
int wc_SlhDsaKey_MakeKey(
    SlhDsaKey * key,
    WC_RNG * rng
)
```

Generates a new SLH-DSA key pair using the RNG for randomness. The key must have been initialized with wc_SlhDsaKey_Init() first.

Parameters:

- **key** Pointer to an initialized SlhDsaKey.
- **rng** Pointer to an initialized WC_RNG.

See:

- wc_SlhDsaKey_Init
- wc_SlhDsaKey_MakeKeyWithRandom

Return:

- 0 on success.
- BAD_FUNC_ARG if key or rng is NULL, or key is not initialized.

Example

```
SlhDsaKey key;
WC_RNG rng;
int ret;

wc_InitRng(&rng);
wc_SlhDsaKey_Init(&key, SLHDSA_SHAKE128F, NULL, INVALID_DEVID);
ret = wc_SlhDsaKey_MakeKey(&key, &rng);
if (ret != 0) {
    // error generating key
}
```

C.67.2.6 function wc_SlhDsaKey_MakeKeyWithRandom

```
int wc_SlhDsaKey_MakeKeyWithRandom(
    SlhDsaKey * key,
    const byte * sk_seed,
    word32 sk_seed_len,
    const byte * sk_prf,
    word32 sk_prf_len,
    const byte * pk_seed,
    word32 pk_seed_len
)
```

Generates an SLH-DSA key pair from caller-provided seed material. This is the deterministic key generation interface — given the same seeds, the same key pair is produced.

Parameters:

- **key** Pointer to an initialized SlhDsaKey.
- **sk_seed** Secret key seed (n bytes).
- **sk_seed_len** Length of sk_seed.
- **sk_prf** Secret key PRF seed (n bytes).
- **sk_prf_len** Length of sk_prf.
- **pk_seed** Public key seed (n bytes).
- **pk_seed_len** Length of pk_seed.

See: wc_SlhDsaKey_MakeKey

Return:

- 0 on success.
- BAD_FUNC_ARG if key or any seed pointer is NULL, or lengths do not match the parameter set's n value.

Example

```
SlhDsaKey key;
byte sk_seed[16], sk_prf[16], pk_seed[16]; // n=16 for 128-bit params
int ret;

// fill seeds with known values (e.g. from NIST test vectors)
wc_SlhDsaKey_Init(&key, SLHDSA_SHAKE128F, NULL, INVALID_DEVID);
ret = wc_SlhDsaKey_MakeKeyWithRandom(&key,
    sk_seed, sizeof(sk_seed),
    sk_prf, sizeof(sk_prf),
    pk_seed, sizeof(pk_seed));
```

C.67.2.7 function wc_SlhDsaKey_SignDeterministic

```
int wc_SlhDsaKey_SignDeterministic(
    SlhDsaKey * key,
    const byte * ctx,
    byte ctxSz,
    const byte * msg,
    word32 msgSz,
    byte * sig,
    word32 * sigSz
)
```

Signs a message using the SLH-DSA external (pure) interface with deterministic randomness. This is FIPS 205 Algorithm 22 with opt_rand set to PK.seed. The message M is wrapped internally as M' = 0x00 || len(ctx) || ctx || M before signing.

Parameters:

- **key** Pointer to a private SlhDsaKey.
- **ctx** Context string for domain separation. May be NULL if ctxSz is 0.
- **ctxSz** Length of the context string (0-255).
- **msg** Pointer to the message to sign. May be NULL if msgSz is 0.
- **msgSz** Length of the message.
- **sig** Buffer to receive the signature.
- **sigSz** On input, size of sig buffer. On output, actual signature length.

See:

- wc_SlhDsaKey_SignWithRandom
- wc_SlhDsaKey_Sign
- wc_SlhDsaKey_Verify

Return:

- 0 on success.
- BAD_FUNC_ARG if key, sig, or sigSz is NULL, or msg is NULL but msgSz is greater than 0.
- BAD_LENGTH_E if the output buffer is too small.

Example

```
SlhDsaKey key;
byte sig[WC_SLHDSA_MAX_SIG_LEN];
word32 sigSz = sizeof(sig);
byte msg[] = "Hello World!";
int ret;

// key already generated via wc_SlhDsaKey_MakeKey()
ret = wc_SlhDsaKey_SignDeterministic(&key, NULL, 0,
    msg, sizeof(msg), sig, &sigSz);
```

C.67.2.8 function wc_SlhDsaKey_SignWithRandom

```
int wc_SlhDsaKey_SignWithRandom(
    SlhDsaKey * key,
    const byte * ctx,
    byte ctxSz,
    const byte * msg,
    word32 msgSz,
    byte * sig,
    word32 * sigSz,
    const byte * addRnd
)
```

Signs a message using the SLH-DSA external (pure) interface with caller-provided additional randomness. This is FIPS 205 Algorithm 22 with an explicit opt_rand value.

Parameters:

- **key** Pointer to a private SlhDsaKey.
- **ctx** Context string. May be NULL if ctxSz is 0.
- **ctxSz** Length of the context string (0-255).
- **msg** Pointer to the message to sign. May be NULL if msgSz is 0.
- **msgSz** Length of the message.
- **sig** Buffer to receive the signature.
- **sigSz** On input, size of sig buffer. On output, actual signature length.
- **addRnd** Additional randomness (n bytes, where n is the parameter set's security parameter).

See:

- wc_SlhDsaKey_SignDeterministic
- wc_SlhDsaKey_Sign

Return:

- 0 on success.

- BAD_FUNC_ARG if key, sig, sigSz, or addRnd is NULL, or msg is NULL but msgSz is greater than 0.

Example

```
SlhDsaKey key;
byte sig[WC_SLHDSA_MAX_SIG_LEN];
word32 sigSz = sizeof(sig);
byte msg[] = "Hello World!";
byte addRnd[16]; // n=16 for 128-bit params
int ret;

wc_RNG_GenerateBlock(&rng, addRnd, sizeof(addRnd));
ret = wc_SlhDsaKey_SignWithRandom(&key, NULL, 0,
    msg, sizeof(msg), sig, &sigSz, addRnd);
```

C.67.2.9 function wc_SlhDsaKey_Sign

```
int wc_SlhDsaKey_Sign(
    SlhDsaKey * key,
    const byte * ctx,
    byte ctxSz,
    const byte * msg,
    word32 msgSz,
    byte * sig,
    word32 * sigSz,
    WC_RNG * rng
)
```

Signs a message using the SLH-DSA external (pure) interface with RNG-provided randomness. This is the general-purpose signing function that uses the WC_RNG for opt_rand.

Parameters:

- **key** Pointer to a private SlhDsaKey.
- **ctx** Context string. May be NULL if ctxSz is 0.
- **ctxSz** Length of the context string (0-255).
- **msg** Pointer to the message to sign. May be NULL if msgSz is 0.
- **msgSz** Length of the message.
- **sig** Buffer to receive the signature.
- **sigSz** On input, size of sig buffer. On output, actual signature length.
- **rng** Pointer to an initialized WC_RNG.

See:

- wc_SlhDsaKey_SignDeterministic
- wc_SlhDsaKey_Verify

Return:

- 0 on success.
- BAD_FUNC_ARG if key, sig, sigSz, or rng is NULL, or msg is NULL but msgSz is greater than 0.

Example

```
SlhDsaKey key;
WC_RNG rng;
byte sig[WC_SLHDSA_MAX_SIG_LEN];
word32 sigSz = sizeof(sig);
byte msg[] = "Hello World!";
```

```
int ret;

ret = wc_SlhDsaKey_Sign(&key, NULL, 0,
    msg, sizeof(msg), sig, &sigSz, &rng);
```

C.67.2.10 function wc_SlhDsaKey_Verify

```
int wc_SlhDsaKey_Verify(
    SlhDsaKey * key,
    const byte * ctx,
    byte ctxSz,
    const byte * msg,
    word32 msgSz,
    const byte * sig,
    word32 sigSz
)
```

Verifies an SLH-DSA signature over a message using the external (pure) interface. This is FIPS 205 Algorithm 24. The message is wrapped internally as M' = 0x00 || len(ctx) || ctx || M before verification.

Parameters:

- **key** Pointer to a public SlhDsaKey.
- **ctx** Context string. May be NULL if ctxSz is 0.
- **ctxSz** Length of the context string (0-255).
- **msg** Pointer to the message to verify. May be NULL if msgSz is 0.
- **msgSz** Length of the message.
- **sig** Pointer to the signature to verify.
- **sigSz** Length of the signature.

See:

- wc_SlhDsaKey_Sign
- wc_SlhDsaKey_SignDeterministic

Return:

- 0 on success (signature valid).
- BAD_FUNC_ARG if key or sig is NULL, or ctx is NULL but ctxSz is greater than 0, or msg is NULL but msgSz is greater than 0.
- BAD_LENGTH_E if sigSz does not match the parameter set's signature length.
- MISSING_KEY if the public key has not been set.
- SIG_VERIFY_E if the signature is invalid.

Example

```
SlhDsaKey key;
byte sig[...]; // previously generated signature
word32 sigSz;
byte msg[] = "Hello World!";
int ret;

ret = wc_SlhDsaKey_Verify(&key, NULL, 0,
    msg, sizeof(msg), sig, sigSz);
if (ret == 0) {
    // signature is valid
}
```

C.67.2.11 function wc_SlhDsaKey_SignMsgDeterministic

```
int wc_SlhDsaKey_SignMsgDeterministic(
    SlhDsaKey * key,
    const byte * mprime,
    word32 mprimeSz,
    byte * sig,
    word32 * sigSz
)
```

Signs using the SLH-DSA internal interface with deterministic randomness. Unlike the external interface, M' is provided directly by the caller — no wrapping is performed. This corresponds to FIPS 205 Algorithm 19 (slh_sign_internal) with opt_rand set to PK.seed.

Parameters:

- **key** Pointer to a private SlhDsaKey.
- **mprime** Pointer to the pre-constructed M' message.
- **mprimeSz** Length of M'.
- **sig** Buffer to receive the signature.
- **sigSz** On input, size of sig buffer. On output, actual signature length.

See:

- wc_SlhDsaKey_SignMsgWithRandom
- wc_SlhDsaKey_VerifyMsg
- wc_SlhDsaKey_SignDeterministic
- wc_SlhDsaKey_SignHashDeterministic

Return:

- 0 on success.
- BAD_FUNC_ARG if key, mprime, sig, or sigSz is NULL.
- BAD_LENGTH_E if sigSz is less than the parameter set's signature length.
- MISSING_KEY if the private key has not been set.

Use this when the ACVP signatureInterface=internal test framework or a protocol layer has already constructed M'. For HashSLH-DSA the caller builds M' as 0x01 || ctxSz || ctx || OID(hashType) || PHM and passes it in here, where PHM is the hash of the application message under hashType.

Example

```
SlhDsaKey key;
byte sig[WC_SLHDSA_MAX_SIG_LEN];
word32 sigSz = sizeof(sig);
byte mprime[] = { ... }; // pre-constructed M'
int ret;

ret = wc_SlhDsaKey_SignMsgDeterministic(&key,
    mprime, sizeof(mprime), sig, &sigSz);
```

C.67.2.12 function wc_SlhDsaKey_SignMsgWithRandom

```
int wc_SlhDsaKey_SignMsgWithRandom(
    SlhDsaKey * key,
    const byte * mprime,
    word32 mprimeSz,
    byte * sig,
    word32 * sigSz,
```

```
    const byte * addRnd
)
```

Signs using the SLH-DSA internal interface with caller-provided additional randomness. M' is provided directly — no wrapping is performed. This corresponds to FIPS 205 Algorithm 19 (slh_sign_internal) with an explicit opt_rand value. See wc_SlhDsaKey_SignMsgDeterministic for the M' layout used by HashSLH-DSA.

Parameters:

- **key** Pointer to a private SlhDsaKey.
- **mprime** Pointer to the pre-constructed M' message.
- **mprimeSz** Length of M'.
- **sig** Buffer to receive the signature.
- **sigSz** On input, size of sig buffer. On output, actual signature length.
- **addRnd** Additional randomness (n bytes).

See:

- wc_SlhDsaKey_SignMsgDeterministic
- wc_SlhDsaKey_VerifyMsg
- wc_SlhDsaKey_SignHashWithRandom

Return:

- 0 on success.
- BAD_FUNC_ARG if key, mprime, sig, sigSz, or addRnd is NULL.
- BAD_LENGTH_E if sigSz is less than the parameter set's signature length.
- MISSING_KEY if the private key has not been set.

Example

```
SlhDsaKey key;
byte sig[WC_SLHDSA_MAX_SIG_LEN];
word32 sigSz = sizeof(sig);
byte mprime[] = { ... };
byte addRnd[16];
int ret;

wc_RNG_GenerateBlock(&rng, addRnd, sizeof(addRnd));
ret = wc_SlhDsaKey_SignMsgWithRandom(&key,
    mprime, sizeof(mprime), sig, &sigSz, addRnd);
```

C.67.2.13 function wc_SlhDsaKey_VerifyMsg

```
int wc_SlhDsaKey_VerifyMsg(
    SlhDsaKey * key,
    const byte * mprime,
    word32 mprimeSz,
    const byte * sig,
    word32 sigSz
)
```

Verifies an SLH-DSA signature using the internal interface. M' is provided directly — no wrapping is performed. This corresponds to FIPS 205 Algorithm 20 (slh_verify_internal).

Parameters:

- **key** Pointer to a public SlhDsaKey.
- **mprime** Pointer to the pre-constructed M' message.

- **mprimeSz** Length of M'.
- **sig** Pointer to the signature to verify.
- **sigSz** Length of the signature.

See:

- wc_SlhDsaKey_SignMsgDeterministic
- wc_SlhDsaKey_Verify
- wc_SlhDsaKey_VerifyHash

Return:

- 0 on success (signature valid).
- BAD_FUNC_ARG if key, mprime, or sig is NULL.
- BAD_LENGTH_E if sigSz does not match the parameter set's signature length.
- MISSING_KEY if the public key has not been set.
- SIG_VERIFY_E if the signature is invalid.

Example

```
SlhDsaKey key;
byte sig[...]; // previously generated signature
word32 sigSz;
byte mprime[] = { ... };
int ret;

ret = wc_SlhDsaKey_VerifyMsg(&key,
    mprime, sizeof(mprime), sig, sigSz);
if (ret == 0) {
    // signature is valid
}
```

C.67.2.14 function wc_SlhDsaKey_SignHashDeterministic

```
int wc_SlhDsaKey_SignHashDeterministic(
    SlhDsaKey * key,
    const byte * ctx,
    byte ctxSz,
    const byte * hash,
    word32 hashSz,
    enum wc_HashType hashType,
    byte * sig,
    word32 * sigSz
)
```

Signs a caller-pre-hashed message digest using the SLH-DSA external (HashSLH-DSA) interface with deterministic randomness, per FIPS 205 Algorithm 23 with the pre-hash domain separator (0x01). The caller must hash the application message with hashType first and pass the digest as hash; this function does NOT hash its input.

Parameters:

- **key** Pointer to a private SlhDsaKey.
- **ctx** Context string. May be NULL if ctxSz is 0.
- **ctxSz** Length of the context string (0-255).
- **hash** Pointer to the pre-hashed message digest. hashSz must equal the digest size for hashType.
- **hashSz** Length of the digest in bytes.
- **hashType** Hash algorithm used for pre-hashing (selects OID). Supported: WC_HASH_TYPE_SHA224, WC_HASH_TYPE_SHA256, WC_HASH_TYPE_SHA384, WC_HASH_TYPE_SHA512, WC_HASH_TYPE_SHA512_224,

WC_HASH_TYPE_SHA512_256, WC_HASH_TYPE_SHAKE128, WC_HASH_TYPE_SHAKE256, WC_HASH_TYPE_SHA3_22
WC_HASH_TYPE_SHA3_256, WC_HASH_TYPE_SHA3_384, WC_HASH_TYPE_SHA3_512.
- **sig** Buffer to receive the signature.
- **sigSz** On input, size of sig buffer. On output, actual signature length.

See:

- wc_SlhDsaKey_SignHashWithRandom
- wc_SlhDsaKey_SignHash
- wc_SlhDsaKey_VerifyHash
- wc_SlhDsaKey_SignMsgDeterministic

Return:

- 0 on success.
- BAD_FUNC_ARG if key, hash, sig, or sigSz is NULL.
- BAD_FUNC_ARG if hashType is WC_HASH_TYPE_NONE, or if the pre-hash gives less collision strength than the parameter set's security category (FIPS 205 sec. 10.2).
- BAD_LENGTH_E if hashSz does not equal the digest size for hashType (32 for SHAKE128, 64 for SHAKE256 per FIPS 205 Section 10.2.2).
- NOT_COMPILED_IN if hashType is not supported in this build.
- MISSING_KEY if the private key has not been set.

Example

```
SlhDsaKey key;
byte sig[WC_SLHDSA_MAX_SIG_LEN];
word32 sigSz = sizeof(sig);
byte msg[] = "Hello World!";
byte digest[WC_SHA256_DIGEST_SIZE];
int ret;

wc_Sha256Hash(msg, sizeof(msg), digest);
ret = wc_SlhDsaKey_SignHashDeterministic(&key, NULL, 0,
    digest, sizeof(digest), WC_HASH_TYPE_SHA256, sig, &sigSz);
```

C.67.2.15 function wc_SlhDsaKey_SignHashWithRandom

```
int wc_SlhDsaKey_SignHashWithRandom(
    SlhDsaKey * key,
    const byte * ctx,
    byte ctxSz,
    const byte * hash,
    word32 hashSz,
    enum wc_HashType hashType,
    byte * sig,
    word32 * sigSz,
    const byte * addRnd
)
```

Signs a caller-pre-hashed message digest using the SLH-DSA external (HashSLH-DSA) interface with caller-provided additional randomness. The caller must hash the application message with hashType first and pass the digest as hash; this function does NOT hash its input.

Parameters:

- **key** Pointer to a private SlhDsaKey.
- **ctx** Context string. May be NULL if ctxSz is 0.
- **ctxSz** Length of the context string (0-255).

- **hash** Pointer to the pre-hashed message digest. hashSz must equal the digest size for hashType.
- **hashSz** Length of the digest in bytes.
- **hashType** Hash algorithm used for pre-hashing (selects OID).
- **sig** Buffer to receive the signature.
- **sigSz** On input, size of sig buffer. On output, actual signature length.
- **addRnd** Additional randomness (n bytes).

See:

- wc_SlhDsaKey_SignHashDeterministic
- wc_SlhDsaKey_VerifyHash
- wc_SlhDsaKey_SignMsgWithRandom

Return:

- 0 on success.
- BAD_FUNC_ARG if key, hash, sig, sigSz, or addRnd is NULL.
- BAD_FUNC_ARG if hashType is WC_HASH_TYPE_NONE, or if the pre-hash gives less collision strength than the parameter set's security category (FIPS 205 sec. 10.2).
- BAD_LENGTH_E if hashSz does not equal the digest size for hashType (32 for SHAKE128, 64 for SHAKE256 per FIPS 205 Section 10.2.2).
- NOT_COMPILED_IN if hashType is not supported in this build.

C.67.2.16 function wc_SlhDsaKey_SignHash

```
int wc_SlhDsaKey_SignHash(
    SlhDsaKey * key,
    const byte * ctx,
    byte ctxSz,
    const byte * hash,
    word32 hashSz,
    enum wc_HashType hashType,
    byte * sig,
    word32 * sigSz,
    WC_RNG * rng
)
```

Signs a caller-pre-hashed message digest using the SLH-DSA external (HashSLH-DSA) interface with RNG-provided randomness. The caller must hash the application message with hashType first and pass the digest as hash; this function does NOT hash its input.

Parameters:

- **key** Pointer to a private SlhDsaKey.
- **ctx** Context string. May be NULL if ctxSz is 0.
- **ctxSz** Length of the context string (0-255).
- **hash** Pointer to the pre-hashed message digest. hashSz must equal the digest size for hashType.
- **hashSz** Length of the digest in bytes.
- **hashType** Hash algorithm used for pre-hashing (selects OID).
- **sig** Buffer to receive the signature.
- **sigSz** On input, size of sig buffer. On output, actual signature length.
- **rng** Pointer to an initialized WC_RNG.

See:

- wc_SlhDsaKey_SignHashDeterministic
- wc_SlhDsaKey_VerifyHash
- wc_SlhDsaKey_SignMsgDeterministic

Return:

- 0 on success.
- BAD_FUNC_ARG if key, hash, sig, sigSz, or rng is NULL.
- BAD_FUNC_ARG if hashType is WC_HASH_TYPE_NONE, or if the pre-hash gives less collision strength than the parameter set's security category (FIPS 205 sec. 10.2).
- BAD_LENGTH_E if hashSz does not equal the digest size for hashType (32 for SHAKE128, 64 for SHAKE256 per FIPS 205 Section 10.2.2).
- NOT_COMPILED_IN if hashType is not supported in this build.

C.67.2.17 function wc_SlhDsaKey_VerifyHash

```
int wc_SlhDsaKey_VerifyHash(
    SlhDsaKey * key,
    const byte * ctx,
    byte ctxSz,
    const byte * hash,
    word32 hashSz,
    enum wc_HashType hashType,
    const byte * sig,
    word32 sigSz
)
```

Verifies an SLH-DSA signature using the external HashSLH-DSA interface (FIPS 205 Algorithm 25). The caller must hash the application message with hashType first and pass the digest as hash; this function does NOT hash its input.

Parameters:

- **key** Pointer to a public SlhDsaKey.
- **ctx** Context string. May be NULL if ctxSz is 0.
- **ctxSz** Length of the context string (0-255).
- **hash** Pointer to the pre-hashed message digest. hashSz must equal the digest size for hashType.
- **hashSz** Length of the digest in bytes.
- **hashType** Hash algorithm used for pre-hashing (selects OID). Must match the hash used during signing.
- **sig** Pointer to the signature to verify.
- **sigSz** Length of the signature.

See:

- wc_SlhDsaKey_SignHashDeterministic
- wc_SlhDsaKey_Verify
- wc_SlhDsaKey_VerifyMsg

Return:

- 0 on success (signature valid).
- BAD_FUNC_ARG if key, hash, or sig is NULL.
- BAD_FUNC_ARG if hashType is WC_HASH_TYPE_NONE, or if the pre-hash gives less collision strength than the parameter set's security category (FIPS 205 sec. 10.2).
- BAD_LENGTH_E if sigSz does not match the parameter set, or if hashSz does not equal the digest size for hashType (32 for SHAKE128, 64 for SHAKE256 per FIPS 205 Section 10.2.2).
- NOT_COMPILED_IN if hashType is not supported in this build.
- MISSING_KEY if the public key has not been set.
- SIG_VERIFY_E if the signature is invalid.

Example

```
SlhDsaKey key;
byte sig[...];
word32 sigSz;
byte msg[] = "Hello World!";
byte digest[WC_SHA256_DIGEST_SIZE];
int ret;

wc_Sha256Hash(msg, sizeof(msg), digest);
ret = wc_SlhDsaKey_VerifyHash(&key, NULL, 0,
    digest, sizeof(digest), WC_HASH_TYPE_SHA256, sig, sigSz);
if (ret == 0) {
    // signature is valid
}
```

C.67.2.18 function wc_SlhDsaKey_ImportPrivate

```
int wc_SlhDsaKey_ImportPrivate(
    SlhDsaKey * key,
    const byte * in,
    word32 inLen
)
```

Imports an SLH-DSA private key from a raw byte buffer. The buffer must contain the full private key (4*n bytes: SK.seed || SK.prf || PK.seed || PK.root). After import, the key can be used for signing.

Parameters:

- **key** Pointer to an initialized SlhDsaKey.
- **in** Buffer containing the raw private key bytes.
- **inLen** Length of the input buffer.

See:

- wc_SlhDsaKey_ExportPrivate
- wc_SlhDsaKey_ImportPublic

Return:

- 0 on success.
- BAD_FUNC_ARG if key or in is NULL, or inLen does not match the expected private key size for the parameter set.

Example

```
SlhDsaKey key;
byte privKey[...]; // 4*n bytes
int ret;

wc_SlhDsaKey_Init(&key, SLHDSA_SHAKE128F, NULL, INVALID_DEVID);
ret = wc_SlhDsaKey_ImportPrivate(&key, privKey, sizeof(privKey));
```

C.67.2.19 function wc_SlhDsaKey_ImportPublic

```
int wc_SlhDsaKey_ImportPublic(
    SlhDsaKey * key,
    const byte * in,
    word32 inLen
)
```

Imports an SLH-DSA public key from a raw byte buffer. The buffer must contain PK.seed || PK.root (2*n bytes). After import, the key can be used for verification.

Parameters:

- **key** Pointer to an initialized SlhDsaKey.
- **in** Buffer containing the raw public key bytes.
- **inLen** Length of the input buffer.

See:

- wc_SlhDsaKey_ExportPublic
- wc_SlhDsaKey_ImportPrivate

Return:

- 0 on success.
- BAD_FUNC_ARG if key or in is NULL, or inLen does not match the expected public key size.

Example

```
SlhDsaKey key;
byte pubKey[...]; // 2*n bytes
int ret;

wc_SlhDsaKey_Init(&key, SLHDSA_SHAKE128F, NULL, INVALID_DEVID);
ret = wc_SlhDsaKey_ImportPublic(&key, pubKey, sizeof(pubKey));
```

C.67.2.20 function wc_SlhDsaKey_CheckKey

```
int wc_SlhDsaKey_CheckKey(
    SlhDsaKey * key
)
```

Checks the consistency of an SLH-DSA key. For a key with both private and public components, verifies that the public key matches the private key.

Parameters:

- **key** Pointer to the SlhDsaKey to check.

See:

- wc_SlhDsaKey_MakeKey
- wc_SlhDsaKey_ImportPrivate

Return:

- 0 on success (key is valid).
- BAD_FUNC_ARG if key is NULL.

C.67.2.21 function wc_SlhDsaKey_ExportPrivate

```
int wc_SlhDsaKey_ExportPrivate(
    SlhDsaKey * key,
    byte * out,
    word32 * outLen
)
```

Exports the private key from an SLH-DSA key object into a raw byte buffer (4*n bytes).

Parameters:

- **key** Pointer to the SlhDsaKey containing a private key.
- **out** Buffer to receive the raw private key bytes.
- **outLen** On input, size of out buffer. On output, bytes written.

See:

- wc_SlhDsaKey_ImportPrivate
- wc_SlhDsaKey_ExportPublic

Return:

- 0 on success.
- BAD_FUNC_ARG if key, out, or outLen is NULL.
- MISSING_KEY if the key object holds no private key.
- BUFFER_E if the output buffer is too small.

Example

```
SlhDsaKey key;
byte privKey[4 * 32]; // 4*n for 256-bit params
word32 privKeySz = sizeof(privKey);
int ret;

ret = wc_SlhDsaKey_ExportPrivate(&key, privKey, &privKeySz);
```

C.67.2.22 function wc_SlhDsaKey_ExportPublic

```
int wc_SlhDsaKey_ExportPublic(
    SlhDsaKey * key,
    byte * out,
    word32 * outLen
)
```

Exports the public key from an SLH-DSA key object into a raw byte buffer (2*n bytes: PK.seed || PK.root).

Parameters:

- **key** Pointer to the SlhDsaKey containing a public key.
- **out** Buffer to receive the raw public key bytes.
- **outLen** On input, size of out buffer. On output, bytes written.

See:

- wc_SlhDsaKey_ImportPublic
- wc_SlhDsaKey_ExportPrivate

Return:

- 0 on success.
- BAD_FUNC_ARG if key, out, or outLen is NULL.
- MISSING_KEY if the key object holds no public key.
- BUFFER_E if the output buffer is too small.

Example

```
SlhDsaKey key;
byte pubKey[2 * 32];
word32 pubKeySz = sizeof(pubKey);
int ret;

ret = wc_SlhDsaKey_ExportPublic(&key, pubKey, &pubKeySz);
```

C.67.2.23 function wc_SlhDsaKey_PrivateSize

```
int wc_SlhDsaKey_PrivateSize(
    SlhDsaKey * key
)
```

Returns the private key size in bytes for the key's parameter set.

Parameters:

- **key** Pointer to an initialized SlhDsaKey.

See:

- wc_SlhDsaKey_PublicSize
- wc_SlhDsaKey_SigSize
- wc_SlhDsaKey_PrivateSizeFromParam

Return:

- Private key size in bytes (4*n) on success.
- BAD_FUNC_ARG if key is NULL or not initialized.

C.67.2.24 function wc_SlhDsaKey_PublicSize

```
int wc_SlhDsaKey_PublicSize(
    SlhDsaKey * key
)
```

Returns the public key size in bytes for the key's parameter set.

Parameters:

- **key** Pointer to an initialized SlhDsaKey.

See:

- wc_SlhDsaKey_PrivateSize
- wc_SlhDsaKey_SigSize
- wc_SlhDsaKey_PublicSizeFromParam

Return:

- Public key size in bytes (2*n) on success.
- BAD_FUNC_ARG if key is NULL or not initialized.

C.67.2.25 function wc_SlhDsaKey_SigSize

```
int wc_SlhDsaKey_SigSize(
    SlhDsaKey * key
)
```

Returns the signature size in bytes for the key's parameter set.

Parameters:

- **key** Pointer to an initialized SlhDsaKey.

See:

- wc_SlhDsaKey_PrivateSize
- wc_SlhDsaKey_PublicSize
- wc_SlhDsaKey_SigSizeFromParam

Return:

- Signature size in bytes on success.
- BAD_FUNC_ARG if key is NULL or not initialized.

C.67.2.26 function wc_SlhDsaKey_PrivateSizeFromParam

```
int wc_SlhDsaKey_PrivateSizeFromParam(
    enum SlhDsaParam param
)
```

Returns the private key size in bytes for the given parameter set without needing an initialized key object.

Parameters:

- **param** The SLH-DSA parameter set.

See: wc_SlhDsaKey_PrivateSize

Return:

- Private key size in bytes (4*n) on success.
- BAD_FUNC_ARG if param is invalid.

C.67.2.27 function wc_SlhDsaKey_PublicSizeFromParam

```
int wc_SlhDsaKey_PublicSizeFromParam(
    enum SlhDsaParam param
)
```

Returns the public key size in bytes for the given parameter set without needing an initialized key object.

Parameters:

- **param** The SLH-DSA parameter set.

See: wc_SlhDsaKey_PublicSize

Return:

- Public key size in bytes (2*n) on success.
- BAD_FUNC_ARG if param is invalid.

C.67.2.28 function wc_SlhDsaKey_SigSizeFromParam

```
int wc_SlhDsaKey_SigSizeFromParam(
    enum SlhDsaParam param
)
```

Returns the signature size in bytes for the given parameter set without needing an initialized key object.

Parameters:

- **param** The SLH-DSA parameter set.

See: wc_SlhDsaKey_SigSize

Return:

- Signature size in bytes on success.
- BAD_FUNC_ARG if param is invalid.

C.67.2.29 function wc_SlhDsaKey_PrivateKeyDecode

```
int wc_SlhDsaKey_PrivateKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    SlhDsaKey * key,
    word32 inSz
)
```

Decodes a DER-encoded SLH-DSA private key in the PKCS#8 OneAsymmetricKey format defined by RFC 9909. The privateKey OCTET STRING contains the raw concatenation SK.seed || SK.prf || PK.seed || PK.root (4*n bytes) directly, without a nested OCTET STRING wrapper as used by Ed25519/Ed448. The SLH-DSA parameter set is detected from the AlgorithmIdentifier OID and key->params is updated to match. Available only when WOLFSSL_SLHDSA_VERIFY_ONLY is not defined.

Parameters:

- **input** DER-encoded key data.
- **inOutIdx** On input, starting offset into input. On output, advanced past the parsed key (unchanged on failure).
- **key** SLH-DSA key. Parameter set is auto-detected from the encoded OID.
- **inSz** Total size of input in bytes.

See:

- wc_SlhDsaKey_KeyToDer
- wc_SlhDsaKey_PublicKeyDecode
- wc_SlhDsaKey_ImportPrivate

Return:

- 0 on success.
- BAD_FUNC_ARG if input, inOutIdx, or key is NULL, or inSz is 0.
- ASN_PARSE_E if the DER cannot be parsed as an SLH-DSA private key (malformed input, wrong key size, or trailing-field violation).
- NOT_COMPILED_IN if the OID names an SLH-DSA variant that is not built into this library.

On a failure that is detected before any write to key->sk (BAD_FUNC_ARG, header/OID parse errors, or wrong privateKey length), the key state is left untouched. On a failure detected after wc_SlhDsaKey_ImportPrivate has populated key->sk (a SHA-2 precompute error, or a trailing-field validation error), key->sk is scrubbed with ForceZero and the WC_SLHDSA_FLAG_PRIVATE/PUBLIC flags are cleared so flags can never claim valid bytes that were zeroed. In both rollback cases, key->params and inOutIdx are restored to their pre-call values.

C.67.2.30 function wc_SlhDsaKey_PublicKeyDecode

```
int wc_SlhDsaKey_PublicKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    SlhDsaKey * key,
    word32 inSz
)
```

Decodes a DER-encoded SLH-DSA public key in the SubjectPublicKeyInfo (SPKI) format. The SLH-DSA parameter set is detected from the AlgorithmIdentifier OID and key->params is updated accordingly.

Parameters:

- **input** DER-encoded key data, or a raw 2*n public key when key->params is already set.

- **inOutIdx** On input, starting offset into input. On output, advanced past the parsed key (unchanged on failure).
- **key** SLH-DSA key. Parameter set is auto-detected from the encoded OID, or honored as-is in the raw fast path.
- **inSz** Total size of input in bytes.

See:

- wc_SlhDsaKey_PublicKeyToDer
- wc_SlhDsaKey_PrivateKeyDecode
- wc_SlhDsaKey_ImportPublic

Return:

- 0 on success.
- BAD_FUNC_ARG if input, inOutIdx, or key is NULL, or inSz is 0.
- ASN_PARSE_E if the DER cannot be parsed as an SLH-DSA public key.
- NOT_COMPILED_IN if the OID names an SLH-DSA variant that is not built into this library.

As a fast path, if key->params is already set the function first hands the entire window from inOutIdx to inSz to wc_SlhDsaKey_ImportPublic. ImportPublic's length check is the disambiguator: a window of exactly 2*n bytes is accepted as a raw public key (PK.seed || PK.root) and consumed in full; any other length is rejected and the function falls through to SPKI parsing. SPKI input always carries enough AlgorithmIdentifier/BIT STRING overhead that it never collides with the 2*n raw length, so it falls through cleanly. The caller does not need to pre-trim the window to 2*n.

On a failure detected before any write (BAD_FUNC_ARG or a malformed SPKI), the key state is left untouched. On a failure detected after ImportPublic has populated the public half of key->sk (a SHA-2 precompute error), the public half sk[2*n .. 4*n] is scrubbed and WC_SLHDSA_FLAG_PUBLIC is cleared from the flags; the private half is left intact in case the caller imported it earlier. key->params and inOutIdx are restored to their pre-call values.

C.67.2.31 function wc_SlhDsaKey_KeyToDer

```
int wc_SlhDsaKey_KeyToDer(
    SlhDsaKey * key,
    byte * output,
    word32 inLen
)
```

Encodes an SLH-DSA private key to DER in the PKCS#8 OneAsymmetricKey format defined by RFC 9909. The privateKey OCTET STRING contains the raw 4*n bytes (SK.seed || SK.prf || PK.seed || PK.root) directly, without the nested OCTET STRING wrapping used by Ed25519/Ed448.

Parameters:

- **key** SLH-DSA key with a populated private key.
- **output** Buffer to receive the DER encoding, or NULL to query the required size.
- **inLen** Size of output in bytes (ignored when output is NULL).

See:

- wc_SlhDsaKey_PrivateKeyDecode
- wc_SlhDsaKey_PrivateKeyToDer
- wc_SlhDsaKey_PublicKeyToDer

Return:

- Size of the encoded DER in bytes on success. Pass NULL as output to query the required buffer size without writing.
- BAD_FUNC_ARG if key or key->params is NULL.

- MISSING_KEY if the private key has not been set.
- BUFFER_E if output is non-NULL and inLen is smaller than the required size.
- NOT_COMPILED_IN if key->params names an SLH-DSA variant whose parameter set is not built in.

Available only when WOLFSSL_SLHDSA_VERIFY_ONLY is not defined and WC_ENABLE_ASYM_KEY_EXPORT is set.

C.67.2.32 function wc_SlhDsaKey_PrivateKeyToDer

```
int wc_SlhDsaKey_PrivateKeyToDer(
    SlhDsaKey * key,
    byte * output,
    word32 inLen
)
```

Encodes an SLH-DSA private key to DER. RFC 9909 packs SK.seed || SK.prf || PK.seed || PK.root into a single OCTET STRING, so SLH-DSA has no distinct private-only encoding. This function is an intentional alias of wc_SlhDsaKey_KeyToDer, kept for API parity with Ed25519/Ed448 which do have a separate private form.

Parameters:

- **key** SLH-DSA key with a populated private key.
- **output** Buffer to receive the DER encoding, or NULL to query the required size.
- **inLen** Size of output in bytes (ignored when output is NULL).

See:

- wc_SlhDsaKey_KeyToDer
- wc_SlhDsaKey_PrivateKeyDecode

Return:

- Size of the encoded DER in bytes on success. Pass NULL as output to query the required buffer size.
- BAD_FUNC_ARG if key or key->params is NULL.
- MISSING_KEY if the private key has not been set.
- BUFFER_E if output is non-NULL and inLen is smaller than the required size.
- NOT_COMPILED_IN if key->params names an SLH-DSA variant whose parameter set is not built in.

Available only when WOLFSSL_SLHDSA_VERIFY_ONLY is not defined and WC_ENABLE_ASYM_KEY_EXPORT is set.

Return codes are inherited unchanged from wc_SlhDsaKey_KeyToDer.

C.67.2.33 function wc_SlhDsaKey_PublicKeyToDer

```
int wc_SlhDsaKey_PublicKeyToDer(
    SlhDsaKey * key,
    byte * output,
    word32 inLen,
    int withAlg
)
```

Encodes an SLH-DSA public key to DER. When withAlg is non-zero the output is a full SubjectPublicKeyInfo structure (AlgorithmIdentifier plus BIT STRING). When withAlg is zero the output contains the raw public key bytes without the SPKI wrapping.

Parameters:

- **key** SLH-DSA key with a populated public key.
- **output** Buffer to receive the DER encoding, or NULL to query the required size.
- **inLen** Size of output in bytes (ignored when output is NULL).
- **withAlg** Non-zero to emit SubjectPublicKeyInfo (with AlgorithmIdentifier); zero to emit the raw public key only.

See:

- wc_SlhDsaKey_PublicKeyDecode
- wc_SlhDsaKey_KeyToDer

Return:

- Size of the encoded DER in bytes on success. Pass NULL as output to query the required buffer size.
- BAD_FUNC_ARG if key or key->params is NULL.
- BUFFER_E if output is non-NULL and inLen is smaller than the required size.
- NOT_COMPILED_IN if key->params names an SLH-DSA variant whose parameter set is not built in.

Available only when WC_ENABLE_ASYM_KEY_EXPORT is set.

C.67.3 Source code

```
int wc_SlhDsaKey_Init(SlhDsaKey* key, enum SlhDsaParam param,
    void* heap, int devId);

int wc_SlhDsaKey_Init_id(SlhDsaKey* key, enum SlhDsaParam param,
    const unsigned char* id, int len, void* heap, int devId);

int wc_SlhDsaKey_Init_label(SlhDsaKey* key, enum SlhDsaParam param,
    const char* label, void* heap, int devId);

void wc_SlhDsaKey_Free(SlhDsaKey* key);

int wc_SlhDsaKey_MakeKey(SlhDsaKey* key, WC_RNG* rng);

int wc_SlhDsaKey_MakeKeyWithRandom(SlhDsaKey* key,
    const byte* sk_seed, word32 sk_seed_len,
    const byte* sk_prf, word32 sk_prf_len,
    const byte* pk_seed, word32 pk_seed_len);

int wc_SlhDsaKey_SignDeterministic(SlhDsaKey* key, const byte* ctx,
    byte ctxSz, const byte* msg, word32 msgSz, byte* sig, word32* sigSz);

int wc_SlhDsaKey_SignWithRandom(SlhDsaKey* key, const byte* ctx,
    byte ctxSz, const byte* msg, word32 msgSz, byte* sig, word32* sigSz,
    const byte* addRnd);

int wc_SlhDsaKey_Sign(SlhDsaKey* key, const byte* ctx,
    byte ctxSz, const byte* msg, word32 msgSz, byte* sig, word32* sigSz,
    WC_RNG* rng);

int wc_SlhDsaKey_Verify(SlhDsaKey* key, const byte* ctx,
```

```
    byte ctxSz, const byte* msg, word32 msgSz, const byte* sig,
    word32 sigSz);

int wc_SlhDsaKey_SignMsgDeterministic(SlhDsaKey* key,
    const byte* mprime, word32 mprimeSz, byte* sig, word32* sigSz);

int wc_SlhDsaKey_SignMsgWithRandom(SlhDsaKey* key,
    const byte* mprime, word32 mprimeSz, byte* sig, word32* sigSz,
    const byte* addRnd);

int wc_SlhDsaKey_VerifyMsg(SlhDsaKey* key, const byte* mprime,
    word32 mprimeSz, const byte* sig, word32 sigSz);

int wc_SlhDsaKey_SignHashDeterministic(SlhDsaKey* key,
    const byte* ctx, byte ctxSz, const byte* hash, word32 hashSz,
    enum wc_HashType hashType, byte* sig, word32* sigSz);

int wc_SlhDsaKey_SignHashWithRandom(SlhDsaKey* key,
    const byte* ctx, byte ctxSz, const byte* hash, word32 hashSz,
    enum wc_HashType hashType, byte* sig, word32* sigSz, const byte* addRnd);

int wc_SlhDsaKey_SignHash(SlhDsaKey* key, const byte* ctx,
    byte ctxSz, const byte* hash, word32 hashSz, enum wc_HashType hashType,
    byte* sig, word32* sigSz, WC_RNG* rng);

int wc_SlhDsaKey_VerifyHash(SlhDsaKey* key, const byte* ctx,
    byte ctxSz, const byte* hash, word32 hashSz, enum wc_HashType hashType,
    const byte* sig, word32 sigSz);

int wc_SlhDsaKey_ImportPrivate(SlhDsaKey* key, const byte* in,
    word32 inLen);

int wc_SlhDsaKey_ImportPublic(SlhDsaKey* key, const byte* in,
    word32 inLen);

int wc_SlhDsaKey_CheckKey(SlhDsaKey* key);

int wc_SlhDsaKey_ExportPrivate(SlhDsaKey* key, byte* out,
    word32* outLen);

int wc_SlhDsaKey_ExportPublic(SlhDsaKey* key, byte* out,
    word32* outLen);

int wc_SlhDsaKey_PrivateSize(SlhDsaKey* key);

int wc_SlhDsaKey_PublicSize(SlhDsaKey* key);

int wc_SlhDsaKey_SigSize(SlhDsaKey* key);

int wc_SlhDsaKey_PrivateSizeFromParam(enum SlhDsaParam param);

int wc_SlhDsaKey_PublicSizeFromParam(enum SlhDsaParam param);

int wc_SlhDsaKey_SigSizeFromParam(enum SlhDsaParam param);
```

```
int wc_SlhDsaKey_PrivateKeyDecode(const byte* input, word32* inOutIdx,
    SlhDsaKey* key, word32 inSz);

int wc_SlhDsaKey_PublicKeyDecode(const byte* input, word32* inOutIdx,
    SlhDsaKey* key, word32 inSz);

int wc_SlhDsaKey_KeyToDer(SlhDsaKey* key, byte* output, word32 inLen);

int wc_SlhDsaKey_PrivateKeyToDer(SlhDsaKey* key, byte* output, word32 inLen);

int wc_SlhDsaKey_PublicKeyToDer(SlhDsaKey* key, byte* output, word32 inLen,
    int withAlg);
```

C.68 wc_xmss.h

C.68.1 Functions

	Name
int	**wc_XmssKey_Init**(XmssKey * key, void * heap, int devId)Initializes an XmssKey object. Must be called before any other XMSS/XMSS^MT operation. Use wc_XmssKey_Free() to release resources when done.
int	**wc_XmssKey_InitId**(XmssKey * key, const unsigned char * id, int len, void * heap, int devId)Initializes an XmssKey with a device_side key identifier. Equivalent to wc_XmssKey_Init() but also stashes a binary id that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.
int	**wc_XmssKey_InitLabel**(XmssKey * key, const char * label, void * heap, int devId)Initializes an XmssKey with a device_side key label. Equivalent to wc_XmssKey_Init() but also stashes a label string that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.
int	**wc_XmssKey_SetParamStr**(XmssKey * key, const char * str)Selects an XMSS or XMSS^MT parameter set by its RFC 8391 name. Accepted names take the form “XMSS_” (single_tree) or “XMSSMT_/” (multi_tree), for example “XMSS-SHA2_10_256” or “XMSSMT-SHA2_20/2_256”. The set of names actually accepted depends on the hash families and tree heights enabled at build time.

	Name
int	**wc_XmssKey_GetParamStr**(const XmssKey * key, const char ** str)Retrieves the parameter set name currently selected on this key. The returned pointer is a static string and must not be freed by the caller.
int	**wc_XmssKey_SetWriteCb**(XmssKey * key, wc_xmss_write_private_key_cb write_cb)Registers the callback that wolfSSL invokes to persist updated private key state. Because XMSS/XMSS^MT is stateful, the application MUST persist the private key after each successful sign before the signature is released; otherwise a crash or restart can lead to one-time key reuse and break the scheme.
int	**wc_XmssKey_SetReadCb**(XmssKey * key, wc_xmss_read_private_key_cb read_cb)Registers the callback that wolfSSL invokes to load persisted private key state. Used by wc_XmssKey_Reload() to bring a saved key back into memory for further signing.
int	**wc_XmssKey_SetContext**(XmssKey * key, void * context)Sets the opaque context pointer passed to both the read and write private-key callbacks.
int	**wc_XmssKey_MakeKey**(XmssKey * key, WC_RNG * rng)Generates a fresh XMSS/XMSS^MT key pair. The parameter set must already be selected via wc_XmssKey_SetParamStr() and the read/write callbacks must be registered. The newly generated private key is persisted via the write callback before the function returns; on success the key transitions to state WC_XMSS_STATE_OK.
int	**wc_XmssKey_Reload**(XmssKey * key)Reloads a previously generated XMSS/XMSS^MT private key from persistent storage using the registered read callback, restoring the key to a state where it can sign further messages. On success the key is in state WC_XMSS_STATE_OK.
int	**wc_XmssKey_GetPrivLen**(const XmssKey * key, word32 * len)Returns the size in bytes of the encoded private key for the parameter set selected on this key.
int	**wc_XmssKey_GetPubLen**(const XmssKey * key, word32 * len)Returns the size in bytes of the XMSS/XMSS^MT public key for the parameter set selected on this key.
int	**wc_XmssKey_GetSigLen**(const XmssKey * key, word32 * len)Returns the signature size in bytes for the parameter set selected on this key.

	Name
int	**wc_XmssKey_Sign**(XmssKey * key, byte * sig, word32 * sigSz, const byte * msg, int msgSz)Signs msg with the XMSS/XMSS^MT private key in key. On entry *sigSz is the size of the sig buffer; on success it is updated to the bytes written.
int	**wc_XmssKey_SigsLeft**(XmssKey * key)Reports whether this key can create another signature. It does not return a count. A key whose one-time keys are used up returns 0 and should be retired.
void	**wc_XmssKey_Free**(XmssKey * key)Releases resources held by an XmssKey. Safe to call with a NULL pointer. After this call the key is in state WC_XMSS_STATE_FREED and must be re-initialized before reuse.
int	**wc_XmssKey_ExportPub**(XmssKey * keyDst, const XmssKey * keySrc)Copies the public part of keySrc into keyDst. The destination key inherits the same parameter set and may be used for verification; it does not carry the private key state and cannot sign. Useful for handing a verifier the minimal data it needs.
int	**wc_XmssKey_ExportPub_ex**(XmssKey * keyDst, const XmssKey * keySrc, void * heap, int devId)Like wc_XmssKey_ExportPub() but the destination key is initialized fresh with the supplied heap and devId.
int	**wc_XmssKey_ExportPubRaw**(const XmssKey * key, byte * out, word32 * outLen)Exports the XMSS/XMSS^MT public key as a raw byte string. On entry *outLen is the size of out; on success it is updated to the bytes written.
int	**wc_XmssKey_ImportPubRaw**(XmssKey * key, const byte * in, word32 inLen)Imports a raw XMSS public key into key. The key must be in state WC_XMSS_STATE_INITED and the parameter set must already be selected (the raw encoding does NOT carry the parameter set, so the caller must apply it via wc_XmssKey_SetParamStr() first). On success the key transitions to state WC_XMSS_STATE_VERIFYONLY.
int	**wc_XmssKey_ImportPubRaw_ex**(XmssKey * key, const byte * in, word32 inLen, int is_xmssmt)Like wc_XmssKey_ImportPubRaw() but explicitly declares whether the encoded key is single_tree XMSS or multi_tree XMSS^MT (pass non_zero for XMSS^MT, zero for XMSS).

	Name
int	**wc_XmssKey_Verify**(XmssKey * key, const byte * sig, word32 sigSz, const byte * msg, int msgSz)Verifies an XMSS/XMSS^MT signature against msg using the public key held in key. The function returns 0 only when the signature is valid; any other value indicates the signature was rejected.

C.68.2 Functions Documentation

C.68.2.1 function wc_XmssKey_Init

```
int wc_XmssKey_Init(
    XmssKey * key,
    void * heap,
    int devId
)
```

Initializes an XmssKey object. Must be called before any other XMSS/XMSS^MT operation. Use wc_XmssKey_Free() to release resources when done.

Parameters:

- **key** Pointer to the XmssKey to initialize.
- **heap** Heap hint for dynamic memory allocation. May be NULL.
- **devId** Device identifier for hardware crypto callbacks. Use INVALID_DEVID for software-only.

See:

- wc_XmssKey_Free
- wc_XmssKey_SetParamStr
- wc_XmssKey_MakeKey

Return:

- 0 on success.
- BAD_FUNC_ARG if key is NULL.

XMSS (eXtended Merkle Signature Scheme) and its multi-tree variant XMSS^MT (RFC 8391, NIST SP 800-208) are STATEFUL hash-based signature schemes: each call to wc_XmssKey_Sign() consumes a one_time component of the private key, and reusing a one_time key destroys the security of the scheme. Applications MUST persist the private key state between sign calls; see wc_XmssKey_SetWriteCb() and wc_XmssKey_SetReadCb().

After init the key is in state WC_XMSS_STATE_INITED. The parameter set must be selected by name with wc_XmssKey_SetParamStr() before generating or reloading a key.

Example

```
XmssKey key;
int ret;

ret = wc_XmssKey_Init(&key, NULL, INVALID_DEVID);
if (ret != 0) {
    // error initializing key
}
wc_XmssKey_SetParamStr(&key, "XMSS-SHA2_10_256");
```

```
// ... use key ...
wc_XmssKey_Free(&key);
```

C.68.2.2 function wc_XmssKey_InitId

```
int wc_XmssKey_InitId(
    XmssKey * key,
    const unsigned char * id,
    int len,
    void * heap,
    int devId
)
```

Initializes an XmssKey with a device-side key identifier. Equivalent to wc_XmssKey_Init() but also stashes a binary id that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.

Parameters:

- **key** Pointer to the XmssKey to initialize.
- **id** Pointer to the device-side key identifier bytes.
- **len** Number of bytes in id; must be in [0, XMSS_MAX_ID_LEN].
- **heap** Heap hint for dynamic memory allocation.
- **devId** Device identifier for the crypto callback.

See:

- wc_XmssKey_Init
- wc_XmssKey_InitLabel
- wc_XmssKey_Free

Return:

- 0 on success.
- BAD_FUNC_ARG if key is NULL, or id is NULL while len > 0.
- BUFFER_E if len is negative or greater than XMSS_MAX_ID_LEN.

The id is copied into the key object; the caller may free its buffer immediately after this call returns.

C.68.2.3 function wc_XmssKey_InitLabel

```
int wc_XmssKey_InitLabel(
    XmssKey * key,
    const char * label,
    void * heap,
    int devId
)
```

Initializes an XmssKey with a device-side key label. Equivalent to wc_XmssKey_Init() but also stashes a label string that a crypto callback can use to look up the underlying key material on the device. Only available when wolfSSL is built with WOLF_PRIVATE_KEY_ID.

Parameters:

- **key** Pointer to the XmssKey to initialize.
- **label** NUL-terminated device-side key label.
- **heap** Heap hint for dynamic memory allocation.
- **devId** Device identifier for the crypto callback.

See:

- wc_XmssKey_Init
- wc_XmssKey_InitId

Return:

- 0 on success.
- BAD_FUNC_ARG if key or label is NULL.
- BUFFER_E if label is empty or longer than XMSS_MAX_LABEL_LEN.

C.68.2.4 function wc_XmssKey_SetParamStr

```
int wc_XmssKey_SetParamStr(
    XmssKey * key,
    const char * str
)
```

Selects an XMSS or XMSS^MT parameter set by its RFC 8391 name. Accepted names take the form "XMSS-" (single-tree) or "XMSSMT-/" (multi-tree), for example "XMSS-SHA2_10_256" or "XMSSMT-SHA2_20/2_256". The set of names actually accepted depends on the hash families and tree heights enabled at build time.

Parameters:

- **key** Pointer to an XmssKey in state WC_XMSS_STATE_INITED.
- **str** Parameter set name (NUL-terminated).

See:

- wc_XmssKey_GetParamStr
- wc_XmssKey_MakeKey

Return:

- 0 on success.
- BAD_FUNC_ARG if key or str is NULL, or if the named parameter set is unknown or not compiled in.
- BAD_STATE_E if key is not in state WC_XMSS_STATE_INITED.

Example

```
XmssKey key;

wc_XmssKey_Init(&key, NULL, INVALID_DEVID);
wc_XmssKey_SetParamStr(&key, "XMSS-SHA2_10_256");
```

C.68.2.5 function wc_XmssKey_GetParamStr

```
int wc_XmssKey_GetParamStr(
    const XmssKey * key,
    const char ** str
)
```

Retrieves the parameter set name currently selected on this key. The returned pointer is a static string and must not be freed by the caller.

Parameters:

- **key** Pointer to an XmssKey with a parameter set selected.
- **str** Receives a pointer to a static parameter name string.

See: wc_XmssKey_SetParamStr

Return:

- 0 on success.
- BAD_FUNC_ARG if key or str is NULL, or no parameter set has been selected.

C.68.2.6 function wc_XmssKey_SetWriteCb

```
int wc_XmssKey_SetWriteCb(
    XmssKey * key,
    wc_xmss_write_private_key_cb write_cb
)
```

Registers the callback that wolfSSL invokes to persist updated private key state. Because XMSS/XMSS^MT is stateful, the application MUST persist the private key after each successful sign before the signature is released; otherwise a crash or restart can lead to one-time key reuse and break the scheme.

Parameters:

- **key** Pointer to an XmssKey.
- **write_cb** Callback invoked to persist the private key.

See:

- wc_XmssKey_SetReadCb
- wc_XmssKey_SetContext
- wc_XmssKey_Sign

Return:

- 0 on success.
- BAD_FUNC_ARG if key or write_cb is NULL.

The callback returns one of the wc_XmssRc codes; WC_XMSS_RC_SAVED_TO_NV_MEMORY signals a durable write.

C.68.2.7 function wc_XmssKey_SetReadCb

```
int wc_XmssKey_SetReadCb(
    XmssKey * key,
    wc_xmss_read_private_key_cb read_cb
)
```

Registers the callback that wolfSSL invokes to load persisted private key state. Used by wc_XmssKey_Reload() to bring a saved key back into memory for further signing.

Parameters:

- **key** Pointer to an XmssKey.
- **read_cb** Callback invoked to load the private key.

See:

- wc_XmssKey_SetWriteCb
- wc_XmssKey_Reload

Return:

- 0 on success.
- BAD_FUNC_ARG if key or read_cb is NULL.

C.68.2.8 function wc_XmssKey_SetContext

```
int wc_XmssKey_SetContext(
    XmssKey * key,
    void * context
)
```

Sets the opaque context pointer passed to both the read and write private-key callbacks.

Parameters:

- **key** Pointer to an XmssKey.
- **context** Application-defined pointer; may be NULL.

See:

- wc_XmssKey_SetReadCb
- wc_XmssKey_SetWriteCb

Return:

- 0 on success.
- BAD_FUNC_ARG if key is NULL.

C.68.2.9 function wc_XmssKey_MakeKey

```
int wc_XmssKey_MakeKey(
    XmssKey * key,
    WC_RNG * rng
)
```

Generates a fresh XMSS/XMSS^MT key pair. The parameter set must already be selected via wc_XmssKey_SetParamStr() and the read/write callbacks must be registered. The newly generated private key is persisted via the write callback before the function returns; on success the key transitions to state WC_XMSS_STATE_OK.

Parameters:

- **key** Pointer to an XmssKey in state WC_XMSS_STATE_PARMSET with callbacks set.
- **rng** Pointer to an initialized WC_RNG.

See:

- wc_XmssKey_Sign
- wc_XmssKey_Reload

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- MEMORY_E on allocation failure.

Key generation can be slow for large tree heights; XMSS^MT variants amortize the cost over multiple smaller trees and generate noticeably faster than equivalent single-tree XMSS parameter sets.

Example

```
XmssKey key;
WC_RNG rng;

wc_XmssKey_Init(&key, NULL, INVALID_DEVID);
wc_XmssKey_SetParamStr(&key, "XMSS-SHA2_10_256");
wc_XmssKey_SetWriteCb(&key, my_write_cb);
```

```
wc_XmssKey_SetReadCb(&key, my_read_cb);
wc_XmssKey_SetContext(&key, &my_storage);
wc_InitRng(&rng);

if (wc_XmssKey_MakeKey(&key, &rng) != 0) {
    // error generating key
}
```

C.68.2.10 function wc_XmssKey_Reload

```
int wc_XmssKey_Reload(
    XmssKey * key
)
```

Reloads a previously generated XMSS/XMSS^MT private key from persistent storage using the registered read callback, restoring the key to a state where it can sign further messages. On success the key is in state WC_XMSS_STATE_OK.

Parameters:

- **key** Pointer to an XmssKey with parameters and read callback set.

See:

- wc_XmssKey_MakeKey
- wc_XmssKey_SetReadCb

Return:

- 0 on success, including the device-backed no-op.
- BAD_FUNC_ARG if any required pointer is NULL.
- WC_XMSS_RC_* mapped error if the read callback fails.
- IO_FAILED_E if the private key could not be read.
- NO_VALID_DEVID with WOLF_CRYPTO_CB_ONLY_XMSS when the key is not device-backed.

The same parameter set selected at key-generation time must be reapplied with wc_XmssKey_SetParamStr() before calling Reload.

With WOLF_CRYPTO_CB, a key on a devId with no read callback is taken to be device-backed and Reload is a no-op. With a read callback the software reload runs as usual, which needs the write callback set too.

WOLF_CRYPTO_CB_ONLY_XMSS removes the software reload, so the read callback is never called and only the device-backed no-op succeeds.

A reloaded key holds no public key: neither arm populates it. Call wc_XmssKey_ImportPubRaw() on a separate key to verify, or keep the public key exported at generation time.

C.68.2.11 function wc_XmssKey_GetPrivLen

```
int wc_XmssKey_GetPrivLen(
    const XmssKey * key,
    word32 * len
)
```

Returns the size in bytes of the encoded private key for the parameter set selected on this key.

Parameters:

- **key** Pointer to an XmssKey with parameters set.
- **len** Receives the private key size in bytes.

See:

- wc_XmssKey_GetPubLen
- wc_XmssKey_GetSigLen

Return:

- 0 on success.
- BAD_FUNC_ARG if key or len is NULL.

C.68.2.12 function wc_XmssKey_GetPubLen

```
int wc_XmssKey_GetPubLen(
    const XmssKey * key,
    word32 * len
)
```

Returns the size in bytes of the XMSS/XMSS^MT public key for the parameter set selected on this key.

Parameters:

- **key** Pointer to an XmssKey with parameters set.
- **len** Receives the public key size in bytes.

See: wc_XmssKey_ExportPubRaw

Return:

- 0 on success.
- BAD_FUNC_ARG if key or len is NULL.

C.68.2.13 function wc_XmssKey_GetSigLen

```
int wc_XmssKey_GetSigLen(
    const XmssKey * key,
    word32 * len
)
```

Returns the signature size in bytes for the parameter set selected on this key.

Parameters:

- **key** Pointer to an XmssKey with parameters set.
- **len** Receives the signature size in bytes.

See: wc_XmssKey_Sign

Return:

- 0 on success.
- BAD_FUNC_ARG if key or len is NULL.

C.68.2.14 function wc_XmssKey_Sign

```
int wc_XmssKey_Sign(
    XmssKey * key,
    byte * sig,
    word32 * sigSz,
    const byte * msg,
    int msgSz
)
```

Signs msg with the XMSS/XMSS^MT private key in key. On entry *sigSz is the size of the sig buffer; on success it is updated to the bytes written.

Parameters:

- **key** Pointer to an XmssKey in state WC_XMSS_STATE_OK.
- **sig** Buffer that receives the signature.
- **sigSz** In: size of sig. Out: bytes written.
- **msg** Message to sign.
- **msgSz** Length of msg in bytes.

See:

- wc_XmssKey_Verify
- wc_XmssKey_SigsLeft
- wc_XmssKey_SetWriteCb

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- BUFFER_E if *sigSz is smaller than the signature size.
- Negative error if all one-time keys have been used.

Each successful sign call consumes a one-time component of the private key. The updated key state is persisted via the registered write callback BEFORE the new signature is returned to the caller. If the write callback fails the sign call fails and the signature is not released. When the supply of one-time keys is exhausted the key transitions to state WC_XMSS_STATE_NOSIGS and further sign attempts fail – query wc_XmssKey_SigsLeft() to detect this condition in advance.

C.68.2.15 function wc_XmssKey_SigsLeft

```
int wc_XmssKey_SigsLeft(
    XmssKey * key
)
```

Reports whether this key can create another signature. It does not return a count. A key whose one-time keys are used up returns 0 and should be retired.

Parameters:

- **key** Pointer to an XmssKey in state WC_XMSS_STATE_OK.

See: wc_XmssKey_Sign

Return:

- 1 if at least one signature remains.
- 0 if the one-time keys are exhausted, if key is NULL or not in state WC_XMSS_STATE_OK, if the read callback cannot load the private key, or if a crypto callback device returns an error. A key bound to a device that handles the request gets the device's answer.

C.68.2.16 function wc_XmssKey_Free

```
void wc_XmssKey_Free(
    XmssKey * key
)
```

Releases resources held by an XmssKey. Safe to call with a NULL pointer. After this call the key is in state WC_XMSS_STATE_FREED and must be re-initialized before reuse.

Parameters:

- **key** Pointer to the XmssKey to free.

See: wc_XmssKey_Init

C.68.2.17 function wc_XmssKey_ExportPub

```
int wc_XmssKey_ExportPub(
    XmssKey * keyDst,
    const XmssKey * keySrc
)
```

Copies the public part of keySrc into keyDst. The destination key inherits the same parameter set and may be used for verification; it does not carry the private key state and cannot sign. Useful for handing a verifier the minimal data it needs.

Parameters:

- **keyDst** Pointer to an initialized destination XmssKey.
- **keySrc** Pointer to an XmssKey with the public key.

See:

- wc_XmssKey_ExportPub_ex
- wc_XmssKey_ExportPubRaw

Return:

- 0 on success.
- BAD_FUNC_ARG if keyDst or keySrc is NULL.
- BAD_STATE_E if keySrc holds no public key.

C.68.2.18 function wc_XmssKey_ExportPub_ex

```
int wc_XmssKey_ExportPub_ex(
    XmssKey * keyDst,
    const XmssKey * keySrc,
    void * heap,
    int devId
)
```

Like wc_XmssKey_ExportPub() but the destination key is initialized fresh with the supplied heap and devId.

Parameters:

- **keyDst** Pointer to an XmssKey to populate.
- **keySrc** Pointer to an XmssKey with the public key.
- **heap** Heap hint for keyDst.
- **devId** Device identifier for keyDst.

See: wc_XmssKey_ExportPub

Return:

- 0 on success.
- BAD_FUNC_ARG if keyDst or keySrc is NULL.
- BAD_STATE_E if keySrc holds no public key.

C.68.2.19 function wc_XmssKey_ExportPubRaw

```
int wc_XmssKey_ExportPubRaw(
    const XmssKey * key,
    byte * out,
    word32 * outLen
)
```

Exports the XMSS/XMSS^MT public key as a raw byte string. On entry *outLen is the size of out; on success it is updated to the bytes written.

Parameters:

- **key** Pointer to an XmssKey.
- **out** Buffer that receives the public key.
- **outLen** In: size of out. Out: bytes written.

See:

- wc_XmssKey_ImportPubRaw
- wc_XmssKey_GetPubLen

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- BAD_STATE_E if key holds no public key.
- BUFFER_E if *outLen is smaller than the public key size.

C.68.2.20 function wc_XmssKey_ImportPubRaw

```
int wc_XmssKey_ImportPubRaw(
    XmssKey * key,
    const byte * in,
    word32 inLen
)
```

Imports a raw XMSS public key into key. The key must be in state WC_XMSS_STATE_INITED and the parameter set must already be selected (the raw encoding does NOT carry the parameter set, so the caller must apply it via wc_XmssKey_SetParamStr() first). On success the key transitions to state WC_XMSS_STATE_VERIFYONLY.

Parameters:

- **key** Pointer to an XmssKey with a parameter set.
- **in** Raw public key bytes.
- **inLen** Length of in in bytes.

See:

- wc_XmssKey_ImportPubRaw_ex
- wc_XmssKey_ExportPubRaw
- wc_XmssKey_Verify

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- BUFFER_E if inLen does not match the expected public key size.

C.68.2.21 function wc_XmssKey_ImportPubRaw_ex

```
int wc_XmssKey_ImportPubRaw_ex(
    XmssKey * key,
    const byte * in,
    word32 inLen,
    int is_xmssmt
)
```

Like wc_XmssKey_ImportPubRaw() but explicitly declares whether the encoded key is single_tree XMSS or multi_tree XMSS^MT (pass non_zero for XMSS^MT, zero for XMSS).

Parameters:

- **key** Pointer to an XmssKey with a parameter set.
- **in** Raw public key bytes.
- **inLen** Length of in in bytes.
- **is_xmssmt** Non-zero if the key is XMSS^MT, zero if plain XMSS.

See: wc_XmssKey_ImportPubRaw

Return:

- 0 on success.
- BAD_FUNC_ARG if any required pointer is NULL.
- BUFFER_E if inLen does not match the expected public key size.

C.68.2.22 function wc_XmssKey_Verify

```
int wc_XmssKey_Verify(
    XmssKey * key,
    const byte * sig,
    word32 sigSz,
    const byte * msg,
    int msgSz
)
```

Verifies an XMSS/XMSS^MT signature against msg using the public key held in key. The function returns 0 only when the signature is valid; any other value indicates the signature was rejected.

Parameters:

- **key** Pointer to an XmssKey with the public key.
- **sig** Signature bytes to verify.
- **sigSz** Length of sig in bytes.
- **msg** Message that was signed.
- **msgSz** Length of msg in bytes.

See:

- wc_XmssKey_Sign
- wc_XmssKey_ImportPubRaw

Return:

- 0 on a valid signature.
- BAD_FUNC_ARG if any required pointer is NULL.
- BAD_STATE_E if key holds no public key.
- SIG_VERIFY_E (or similar) if the signature is invalid or malformed.

C.68.3 Source code

```
int wc_XmssKey_Init(XmssKey* key, void* heap, int devId);

int wc_XmssKey_InitId(XmssKey* key, const unsigned char* id, int len,
    void* heap, int devId);

int wc_XmssKey_InitLabel(XmssKey* key, const char* label, void* heap,
    int devId);

int wc_XmssKey_SetParamStr(XmssKey* key, const char* str);

int wc_XmssKey_GetParamStr(const XmssKey* key, const char** str);

int wc_XmssKey_SetWriteCb(XmssKey* key, wc_xmss_write_private_key_cb write_cb);

int wc_XmssKey_SetReadCb(XmssKey* key, wc_xmss_read_private_key_cb read_cb);

int wc_XmssKey_SetContext(XmssKey* key, void* context);

int wc_XmssKey_MakeKey(XmssKey* key, WC_RNG* rng);

int wc_XmssKey_Reload(XmssKey* key);

int wc_XmssKey_GetPrivLen(const XmssKey* key, word32* len);

int wc_XmssKey_GetPubLen(const XmssKey* key, word32* len);

int wc_XmssKey_GetSigLen(const XmssKey* key, word32* len);

int wc_XmssKey_Sign(XmssKey* key, byte* sig, word32* sigSz, const byte* msg,
    int msgSz);

int wc_XmssKey_SigsLeft(XmssKey* key);

void wc_XmssKey_Free(XmssKey* key);

int wc_XmssKey_ExportPub(XmssKey* keyDst, const XmssKey* keySrc);

int wc_XmssKey_ExportPub_ex(XmssKey* keyDst, const XmssKey* keySrc,
    void* heap, int devId);

int wc_XmssKey_ExportPubRaw(const XmssKey* key, byte* out, word32* outLen);

int wc_XmssKey_ImportPubRaw(XmssKey* key, const byte* in, word32 inLen);

int wc_XmssKey_ImportPubRaw_ex(XmssKey* key, const byte* in, word32 inLen,
    int is_xmssmt);

int wc_XmssKey_Verify(XmssKey* key, const byte* sig, word32 sigSz,
    const byte* msg, int msgSz);
```

C.69 wolfio.h

C.69.1 Functions

	Name
int	**EmbedReceive**(WOLFSSL * ssl, char * buf, int sz, void * ctx)This function is the receive embedded callback.
int	**EmbedSend**(WOLFSSL * ssl, char * buf, int sz, void * ctx)This function is the send embedded callback.
int	**EmbedReceiveFrom**(WOLFSSL * ssl, char * buf, int sz, void * ctx)This function is the receive embedded callback.
int	**EmbedSendTo**(WOLFSSL * ssl, char * buf, int sz, void * ctx)This function is the send embedded callback.
int	**EmbedGenerateCookie**(WOLFSSL * ssl, byte * buf, int sz, void * ctx)This function is the DTLS Generate Cookie callback.
void	**EmbedOcspRespFree**(void * ctx, byte * resp)This function frees the response buffer.
void	**wolfSSL_CTX_SetIORecv**(WOLFSSL_CTX * ctx, CallbackIORecv CBIORecv)This function registers a receive callback for wolfSSL to get input data. By default, wolfSSL uses EmbedReceive() as the callback which uses the system's TCP recv() function. The user can register a function to get input from memory, some other network module, or from anywhere. Please see the EmbedReceive() function in src/io.c as a guide for how the function should work and for error codes. In particular, IO_ERR_WANT_READ should be returned for non blocking receive when no data is ready.
void	**wolfSSL_SetIOReadCtx**(WOLFSSL * ssl, void * ctx)This function registers a context for the SSL session's receive callback function. By default, wolfSSL sets the file descriptor passed to wolfSSL_set_fd() as the context when wolfSSL is using the system's TCP library. If you've registered your own receive callback you may want to set a specific context for the session. For example, if you're using memory buffers the context may be a pointer to a structure describing where and how to access the memory buffers.

	Name
void	**wolfSSL_SetIOWriteCtx**(WOLFSSL * ssl, void * ctx)This function registers a context for the SSL session's send callback function. By default, wolfSSL sets the file descriptor passed to wolfSSL_set_fd() as the context when wolfSSL is using the system's TCP library. If you've registered your own send callback you may want to set a specific context for the session. For example, if you're using memory buffers the context may be a pointer to a structure describing where and how to access the memory buffers.
void *	**wolfSSL_GetIOReadCtx**(WOLFSSL * ssl)This function returns the IOCB_ReadCtx member of the WOLFSSL struct.
void *	**wolfSSL_GetIOWriteCtx**(WOLFSSL * ssl)This function returns the IOCB_WriteCtx member of the WOLFSSL structure.
void	**wolfSSL_SetIOReadFlags**(WOLFSSL * ssl, int flags)This function sets the flags for the receive callback to use for the given SSL session. The receive callback could be either the default wolfSSL EmbedReceive callback, or a custom callback specified by the user (see wolfSSL_CTX_SetIORecv). The default flag value is set internally by wolfSSL to the value of 0. The default wolfSSL receive callback uses the recv() function to receive data from the socket. From the recv() man page: "The flags argument to a recv() function is formed by or'ing one or more of the values: MSG_OOB process out-of-band data, MSG_PEEK peek at incoming message, MSG_WAITALL wait for full request or error. The MSG_OOB flag requests receipt of out-of-band data that would not be received in the normal data stream. Some protocols place expedited data at the head of the normal data queue, and thus this flag cannot be used with such protocols. The MSG_PEEK flag causes the receive operation to return data from the beginning of the receive queue without removing that data from the queue. Thus, a subsequent receive call will return the same data. The MSG_WAITALL flag requests that the operation block until the full request is satisfied. However, the call may still return less data than requested if a signal is caught, an error or disconnect occurs, or the next data to be received is of a different type than that returned."

	Name
void	**wolfSSL_SetIOWriteFlags**(WOLFSSL * ssl, int flags)This function sets the flags for the send callback to use for the given SSL session. The send callback could be either the default wolfSSL EmbedSend callback, or a custom callback specified by the user (see wolfSSL_CTX_SetIOSend). The default flag value is set internally by wolfSSL to the value of 0. The default wolfSSL send callback uses the send() function to send data from the socket. From the send() man page: “The flags parameter may include one or more of the following: #define MSG_OOB 0x1 // process out_of_band data, #define MSG_DONTROUTE 0x4 // bypass routing, use direct interface. The flag MSG_OOB is used to send ‘out_of_band’ data on sockets that support this notion (e.g. SOCK_STREAM); the underlying protocol must also support ‘out-of-band’ data. MSG_DONTROUTE is usually used only by diagnostic or routing programs.”
void	**wolfSSL_SetIO_NetX**(WOLFSSL * ssl, NX_TCP_SOCKET * nxsocket, ULONG waitoption)This function sets the nxTcpSocket and nxWait members of the nxCtx struct within the WOLFSSL structure.
void	**wolfSSL_SetIO_NetX_Dtls**(WOLFSSL * ssl, NX_UDP_SOCKET * nxsocket, NXD_ADDRESS nxdip, USHORT nxport, ULONG waitoption)This function configures the NetX Duo UDP context for a DTLS session. It stores the UDP socket, destination IP address (by value), destination port, and wait option into the WOLFSSL nxCtx structure. Requires WOLFSSL_NETX_DUO to be defined (ThreadX NetX Duo SDK).
void	**wolfSSL_CTX_SetGenCookie**(WOLFSSL_CTX * ctx, CallbackGenCookie cb)This function sets the callback for the CBIOCookie member of the WOLFSSL_CTX structure. The CallbackGenCookie type is a function pointer and has the signature: int (*CallbackGenCookie)(WOLFSSL* ssl, unsigned char* buf, int sz, void* ctx);.
void *	**wolfSSL_GetCookieCtx**(WOLFSSL * ssl)This function returns the IOCB_CookieCtx member of the WOLFSSL structure.

	Name
int	**wolfSSL_SetIO_ISOTP**(WOLFSSL * ssl, isotp_wolfssl_ctx * ctx, can_recv_fn recv_fn, can_send_fn send_fn, can_delay_fn delay_fn, word32 receive_delay, char * receive_buffer, int receive_buffer_size, void * arg)This function sets up the ISO-TP context if wolfSSL, for use when wolfSSL is compiled with WOLFSSL_ISOTP.
void	**wolfSSL_SSLDisableRead**(WOLFSSL * ssl)This function disables reading from the IO layer.
void	**wolfSSL_SSLEnableRead**(WOLFSSL * ssl)This function enables reading from the IO layer. Reading is enabled by default and should be used to undo wolfSSL_SSLDisableRead();.
WOLFSSL_API void	**wolfSSL_SetRecvFrom**(WOLFSSL * ssl, WolfSSLRecvFrom recvFrom)Set a custom DTLS recvfrom callback for a WOLFSSL session.
WOLFSSL_API void	**wolfSSL_SetSendTo**(WOLFSSL * ssl, WolfSSLSento sendTo)Set a custom DTLS sendto callback for a WOLFSSL session.
int	**wolfIO_Select**(SOCKET_T sockfd, int to_sec)Waits for socket to be ready for I/O with timeout.
int	**wolfIO_TcpConnect**(SOCKET_T * sockfd, const char * ip, unsigned short port, int to_sec)Connects to TCP server with timeout.
int	**wolfIO_TcpAccept**(SOCKET_T sockfd, SOCKADDR * peer_addr, XSOCKLENT * peer_len)Accepts TCP connection.
int	**wolfIO_TcpBind**(SOCKET_T * sockfd, word16 port)Binds TCP socket to port.
int	**wolfIO_Send**(SOCKET_T sd, char * buf, int sz, int wrFlags)Sends data on socket.
int	**wolfIO_Recv**(SOCKET_T sd, char * buf, int sz, int rdFlags)Receives data from socket.
int	**wolfIO_SendTo**(SOCKET_T sd, WOLFSSL_BIO_ADDR * addr, char * buf, int sz, int wrFlags)Sends datagram to address.
int	**wolfIO_RecvFrom**(SOCKET_T sd, WOLFSSL_BIO_ADDR * addr, char * buf, int sz, int rdFlags)Receives datagram from address.
int	**wolfSSL_BioSend**(WOLFSSL * ssl, char * buf, int sz, void * ctx)BIO send callback.
int	**wolfSSL_BioReceive**(WOLFSSL * ssl, char * buf, int sz, void * ctx)BIO receive callback.
int	**EmbedReceiveFromMcast**(WOLFSSL * ssl, char * buf, int sz, void * ctx)Receives multicast datagram.
int	**wolfIO_HttpBuildRequestOcsp**(const char * domainName, const char * path, int ocspReqSz, unsigned char * buf, int bufSize)Builds HTTP OCSP request.

	Name
int	**wolfIO_HttpProcessResponseOcspGenericIO**(WolfSSLGenericI ioCb, void * ioCbCtx, unsigned char ** respBuf, unsigned char * httpBuf, int httpBufSz, void * heap)Processes HTTP OCSP response with generic I/O.
int	**wolfIO_HttpProcessResponseOcsp**(int sfd, unsigned char ** respBuf, unsigned char * httpBuf, int httpBufSz, void * heap)Processes HTTP OCSP response.
int	**EmbedOcspLookup**(void * ctx, const char * url, int urlSz, byte * ocspReqBuf, int ocspReqSz, byte ** ocspRespBuf)OCSP lookup callback.
int	**wolfIO_HttpBuildRequestCrl**(const char * url, int urlSz, const char * domainName, unsigned char * buf, int bufSize)Builds HTTP CRL request.
int	**wolfIO_HttpProcessResponseCrl**(WOLFSSL_CRL * crl, int sfd, unsigned char * httpBuf, int httpBufSz)Processes HTTP CRL response.
int	**EmbedCrlLookup**(WOLFSSL_CRL * crl, const char * url, int urlSz)CRL lookup callback.
int	**wolfIO_DecodeUrl**(const char * url, int urlSz, char * outName, char * outPath, unsigned short * outPort)Decodes URL into components.
int	**wolfIO_HttpBuildRequest**(const char * reqType, const char * domainName, const char * path, int pathLen, int reqSz, const char * contentType, unsigned char * buf, int bufSize)Builds generic HTTP request.
int	**wolfIO_HttpProcessResponseGenericIO**(WolfSSLGenericIORecv ioCb, void * ioCbCtx, const char ** appStrList, unsigned char ** respBuf, unsigned char * httpBuf, int httpBufSz, int dynType, void * heap)Processes HTTP response with generic I/O.
int	**wolfIO_HttpProcessResponse**(int sfd, const char ** appStrList, unsigned char ** respBuf, unsigned char * httpBuf, int httpBufSz, int dynType, void * heap)Processes HTTP response.
void	**wolfSSL_CTX_SetIOSend**(WOLFSSL_CTX * ctx, CallbackIOSend CBIOSend)Sets I/O send callback for context.
void	**wolfSSL_SSLSetIORecv**(WOLFSSL * ssl, CallbackIORecv CBIORecv)Sets I/O receive callback for SSL object.
void	**wolfSSL_SSLSetIOSend**(WOLFSSL * ssl, CallbackIOSend CBIOSend)Sets I/O send callback for SSL object.
void	**wolfSSL_SetIO_Mynewt**(WOLFSSL * ssl, struct mn_socket * mnSocket, struct mn_sockaddr_in * mnSockAddrIn)Sets I/O for Mynewt platform.

	Name
int	**wolfSSL_SetIO_LwIP**(WOLFSSL * ssl, void * pcb, tcp_recv_fn recv, tcp_sent_fn sent, void * arg)Sets I/O for LwIP platform.
void	**wolfSSL_SetCookieCtx**(WOLFSSL * ssl, void * ctx)Sets cookie context for DTLS.
void	**wolfSSL_CTX_SetIOGetPeer**(WOLFSSL_CTX * ctx, CallbackGetPeer cb)Sets get peer callback for context.
void	**wolfSSL_CTX_SetIOSetPeer**(WOLFSSL_CTX * ctx, CallbackSetPeer cb)Sets set peer callback for context.
int	**EmbedGetPeer**(WOLFSSL * ssl, char * ip, int * ipSz, unsigned short * port, int * fam)Gets peer information.
int	**EmbedSetPeer**(WOLFSSL * ssl, char * ip, int ipSz, unsigned short port, int fam)Sets peer information.

C.69.2 Functions Documentation

C.69.2.1 function EmbedReceive

```
int EmbedReceive(
    WOLFSSL * ssl,
    char * buf,
    int sz,
    void * ctx
)
```

This function is the receive embedded callback.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **buf** a char pointer representation of the buffer.
- **sz** the size of the buffer.
- **ctx** a void pointer to user registered context. In the default case the ctx is a socket descriptor pointer.

See:

- EmbedSend
- wolfSSL_CTX_SetIORecv
- wolfSSL_SSLSetIORecv

Return:

- Success This function returns the number of bytes read.
- WOLFSSL_CBIO_ERR_WANT_READ returned with a “Would block” message if the last error was SOCKET_EWOULDBLCOK or SOCKET_EAGAIN.
- WOLFSSL_CBIO_ERR_TIMEOUT returned with a “Socket timeout” message.
- WOLFSSL_CBIO_ERR_CONN_RST returned with a “Connection reset” message if the last error was SOCKET_ECONNRESET.
- WOLFSSL_CBIO_ERR_ISR returned with a “Socket interrupted” message if the last error was SOCKET_EINTR.

- WOLFSSL_CBIO_ERR_WANT_READ returned with a “Connection refused” message if the last error was SOCKET_ECONNREFUSED.
- WOLFSSL_CBIO_ERR_CONN_CLOSE returned with a “Connection aborted” message if the last error was SOCKET_ECONNABORTED.
- WOLFSSL_CBIO_ERR_GENERAL returned with a “General error” message if the last error was not specified.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
char* buf;
int sz;
void* ctx;
int bytesRead = EmbedReceive(ssl, buf, sz, ctx);
if(bytesRead <= 0){
    // There were no bytes read. Failure case.
}
```

C.69.2.2 function EmbedSend

```
int EmbedSend(
    WOLFSSL * ssl,
    char * buf,
    int sz,
    void * ctx
)
```

This function is the send embedded callback.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **buf** a char pointer representing the buffer.
- **sz** the size of the buffer.
- **ctx** a void pointer to user registered context.

See:

- EmbedReceive
- wolfSSL_CTX_SetIOSend
- wolfSSL_SSLSetIOSend

Return:

- Success This function returns the number of bytes sent.
- WOLFSSL_CBIO_ERR_WANT_WRITE returned with a “Would block” message if the last error was SOCKET_EWOULDBLOCK or SOCKET_EAGAIN.
- WOLFSSL_CBIO_ERR_CONN_RST returned with a “Connection reset” message if the last error was SOCKET_ECONNRESET.
- WOLFSSL_CBIO_ERR_ISR returned with a “Socket interrupted” message if the last error was SOCKET_EINTR.
- WOLFSSL_CBIO_ERR_CONN_CLOSE returned with a “Socket EPIPE” message if the last error was SOCKET_EPIPE.
- WOLFSSL_CBIO_ERR_GENERAL returned with a “General error” message if the last error was not specified.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
char* buf;
int sz;
void* ctx;
int dSent = EmbedSend(ssl, buf, sz, ctx);
if(dSent <= 0){
    // No byes sent. Failure case.
}
```

C.69.2.3 function EmbedReceiveFrom

```
int EmbedReceiveFrom(
    WOLFSSL * ssl,
    char * buf,
    int sz,
    void * ctx
)
```

This function is the receive embedded callback.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **buf** a constant char pointer to the buffer.
- **sz** an int type representing the size of the buffer.
- **ctx** a void pointer to the WOLFSSL_CTX context.

See:

- EmbedSendTo
- wolfSSL_CTX_SetIORecv
- wolfSSL_SSLSetIORecv
- wolfSSL_dtls_get_current_timeout

Return:

- Success This function returns the nb bytes read if the execution was successful.
- WOLFSSL_CBIO_ERR_WANT_READ if the connection refused or if a 'would block' error was thrown in the function.
- WOLFSSL_CBIO_ERR_TIMEOUT returned if the socket timed out.
- WOLFSSL_CBIO_ERR_CONN_RST returned if the connection reset.
- WOLFSSL_CBIO_ERR_ISR returned if the socket was interrupted.
- WOLFSSL_CBIO_ERR_GENERAL returned if there was a general error.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = WOLFSSL_new(ctx);
char* buf;
int sz = sizeof(buf)/sizeof(char);
(void*)ctx;
…
int nb = EmbedReceiveFrom(ssl, buf, sz, ctx);
if(nb > 0){
    // nb is the number of bytes written and is positive
}
```

C.69.2.4 function EmbedSendTo

```
int EmbedSendTo(
    WOLFSSL * ssl,
    char * buf,
    int sz,
    void * ctx
)
```

This function is the send embedded callback.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **buf** a char pointer representing the buffer.
- **sz** the size of the buffer.
- **ctx** a void pointer to the user registered context. The default case is a WOLFSSL_DTLS_CTX structure.

See:

- EmbedReceiveFrom
- wolfSSL_CTX_SetIOSend
- wolfSSL_SSLSetIOSend

Return:

- Success This function returns the number of bytes sent.
- WOLFSSL_CBIO_ERR_WANT_WRITE returned with a "Would Block" message if the last error was either SOCKET_EWOULDBLOCK or SOCKET_EAGAIN error.
- WOLFSSL_CBIO_ERR_CONN_RST returned with a "Connection reset" message if the last error was SOCKET_ECONNRESET.
- WOLFSSL_CBIO_ERR_ISR returned with a "Socket interrupted" message if the last error was SOCKET_EINTR.
- WOLFSSL_CBIO_ERR_CONN_CLOSE returned with a "Socket EPIPE" message if the last error was WOLFSSL_CBIO_ERR_CONN_CLOSE.
- WOLFSSL_CBIO_ERR_GENERAL returned with a "General error" message if the last error was not specified.

Example

```
WOLFSSL* ssl;
…
char* buf;
int sz;
void* ctx;

int sEmbed = EmbedSendto(ssl, buf, sz, ctx);
if(sEmbed <= 0){
    // No bytes sent. Failure case.
}
```

C.69.2.5 function EmbedGenerateCookie

```
int EmbedGenerateCookie(
    WOLFSSL * ssl,
    byte * buf,
    int sz,
```

```
    void * ctx
)
```

This function is the DTLS Generate Cookie callback.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **buf** byte pointer representing the buffer. It is the destination from XMEMCPY().
- **sz** the size of the buffer.
- **ctx** a void pointer to user registered context.

See: wolfSSL_CTX_SetGenCookie

Return:

- Success This function returns the number of bytes copied into the buffer.
- GEN_COOKIE_E returned if the getpeername failed in EmbedGenerateCookie.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
byte buffer[BUFFER_SIZE];
int sz = sizeof(buffer)/sizeof(byte);
void* ctx;
…
int ret = EmbedGenerateCookie(ssl, buffer, sz, ctx);

if(ret > 0){
    // EmbedGenerateCookie code block for success
}
```

C.69.2.6 function EmbedOcspRespFree

```
void EmbedOcspRespFree(
    void * ctx,
    byte * resp
)
```

This function frees the response buffer.

Parameters:

- **ctx** a void pointer to heap hint.
- **resp** a byte pointer representing the response.

See:

- wolfSSL_CertManagerSetOCSP_Cb
- wolfSSL_CertManagerEnableOCSPStapling
- wolfSSL_CertManagerEnableOCSP

Return: none No returns.

Example

```
void* ctx;
byte* resp; // Response buffer.
…
EmbedOcspRespFree(ctx, resp);
```

C.69.2.7 function wolfSSL_CTX_SetIORecv

```
void wolfSSL_CTX_SetIORecv(
    WOLFSSL_CTX * ctx,
    CallbackIORecv CBIORecv
)
```

This function registers a receive callback for wolfSSL to get input data. By default, wolfSSL uses EmbedReceive() as the callback which uses the system's TCP recv() function. The user can register a function to get input from memory, some other network module, or from anywhere. Please see the EmbedReceive() function in src/io.c as a guide for how the function should work and for error codes. In particular, IO_ERR_WANT_READ should be returned for non blocking receive when no data is ready.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **callback** function to be registered as the receive callback for the wolfSSL context, ctx. The signature of this function must follow that as shown above in the Synopsis section.

See:

- wolfSSL_CTX_SetIOSend
- wolfSSL_SetIOReadCtx
- wolfSSL_SetIOWriteCtx

Return: none no Returns.

Example

```
WOLFSSL_CTX* ctx = 0;
// Receive callback prototype
int MyEmbedReceive(WOLFSSL* ssl, char* buf, int sz, void* ctx);
// Register the custom receive callback with wolfSSL
wolfSSL_CTX_SetIORecv(ctx, MyEmbedReceive);
int MyEmbedReceive(WOLFSSL* ssl, char* buf, int sz, void* ctx)
{
    // custom EmbedReceive function
}
```

C.69.2.8 function wolfSSL_SetIOReadCtx

```
void wolfSSL_SetIOReadCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

This function registers a context for the SSL session's receive callback function. By default, wolfSSL sets the file descriptor passed to wolfSSL_set_fd() as the context when wolfSSL is using the system's TCP library. If you've registered your own receive callback you may want to set a specific context for the session. For example, if you're using memory buffers the context may be a pointer to a structure describing where and how to access the memory buffers.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **rctx** pointer to the context to be registered with the SSL session's (ssl) receive callback function.

See:

- wolfSSL_CTX_SetIORecv
- wolfSSL_CTX_SetIOSend

- wolfSSL_SetIOWriteCtx

Return: none No returns.

Example

```
int sockfd;
WOLFSSL* ssl = 0;
...
// Manually setting the socket fd as the receive CTX, for example
wolfSSL_SetIOReadCtx(ssl, &sockfd);
...
```

C.69.2.9 function wolfSSL_SetIOWriteCtx

```
void wolfSSL_SetIOWriteCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

This function registers a context for the SSL session's send callback function. By default, wolfSSL sets the file descriptor passed to wolfSSL_set_fd() as the context when wolfSSL is using the system's TCP library. If you've registered your own send callback you may want to set a specific context for the session. For example, if you're using memory buffers the context may be a pointer to a structure describing where and how to access the memory buffers.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **wctx** pointer to the context to be registered with the SSL session's (ssl) send callback function.

See:

- wolfSSL_CTX_SetIORecv
- wolfSSL_CTX_SetIOSend
- wolfSSL_SetIOReadCtx

Return: none No returns.

Example

```
int sockfd;
WOLFSSL* ssl = 0;
...
// Manually setting the socket fd as the send CTX, for example
wolfSSL_SetIOWriteCtx(ssl, &sockfd);
...
```

C.69.2.10 function wolfSSL_GetIOReadCtx

```
void * wolfSSL_GetIOReadCtx(
    WOLFSSL * ssl
)
```

This function returns the IOCB_ReadCtx member of the WOLFSSL struct.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_GetIOWriteCtx
- wolfSSL_SetIOReadFlags
- wolfSSL_SetIOWriteCtx
- wolfSSL_SetIOReadCtx
- wolfSSL_CTX_SetIOSend

Return:

- pointer This function returns a void pointer to the IOCB_ReadCtx member of the WOLFSSL structure.
- NULL returned if the WOLFSSL struct is NULL.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
void* ioRead;
...
ioRead = wolfSSL_GetIOReadCtx(ssl);
if(ioRead == NULL){
    // Failure case. The ssl object was NULL.
}
```

C.69.2.11 function wolfSSL_GetIOWriteCtx

```
void * wolfSSL_GetIOWriteCtx(
    WOLFSSL * ssl
)
```

This function returns the IOCB_WriteCtx member of the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_GetIOReadCtx
- wolfSSL_SetIOWriteCtx
- wolfSSL_SetIOReadCtx
- wolfSSL_CTX_SetIOSend

Return:

- pointer This function returns a void pointer to the IOCB_WriteCtx member of the WOLFSSL structure.
- NULL returned if the WOLFSSL struct is NULL.

Example

```
WOLFSSL* ssl;
void* ioWrite;
...
ioWrite = wolfSSL_GetIOWriteCtx(ssl);
if(ioWrite == NULL){
    // The function returned NULL.
}
```

C.69.2.12 function wolfSSL_SetIOReadFlags

```
void wolfSSL_SetIOReadFlags(
    WOLFSSL * ssl,
```

```
    int flags
)
```

This function sets the flags for the receive callback to use for the given SSL session. The receive callback could be either the default wolfSSL EmbedReceive callback, or a custom callback specified by the user (see wolfSSL_CTX_SetIORecv). The default flag value is set internally by wolfSSL to the value of 0. The default wolfSSL receive callback uses the recv() function to receive data from the socket. From the recv() man page: "The flags argument to a recv() function is formed by or'ing one or more of the values: MSG_OOB process out-of-band data, MSG_PEEK peek at incoming message, MSG_WAITALL wait for full request or error. The MSG_OOB flag requests receipt of out-of-band data that would not be received in the normal data stream. Some protocols place expedited data at the head of the normal data queue, and thus this flag cannot be used with such protocols. The MSG_PEEK flag causes the receive operation to return data from the beginning of the receive queue without removing that data from the queue. Thus, a subsequent receive call will return the same data. The MSG_WAITALL flag requests that the operation block until the full request is satisfied. However, the call may still return less data than requested if a signal is caught, an error or disconnect occurs, or the next data to be received is of a different type than that returned."

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **flags** value of the I/O read flags for the specified SSL session (ssl).

See:

- wolfSSL_CTX_SetIORecv
- wolfSSL_CTX_SetIOSend
- wolfSSL_SetIOReadCtx

Return: none No returns.

Example

```
WOLFSSL* ssl = 0;
...
// Manually setting recv flags to 0
wolfSSL_SetIOReadFlags(ssl, 0);
...
```

C.69.2.13 function wolfSSL_SetIOWriteFlags

```
void wolfSSL_SetIOWriteFlags(
    WOLFSSL * ssl,
    int flags
)
```

This function sets the flags for the send callback to use for the given SSL session. The send callback could be either the default wolfSSL EmbedSend callback, or a custom callback specified by the user (see wolfSSL_CTX_SetIOSend). The default flag value is set internally by wolfSSL to the value of 0. The default wolfSSL send callback uses the send() function to send data from the socket. From the send() man page: "The flags parameter may include one or more of the following: #define MSG_OOB 0x1 // process out-of-band data, #define MSG_DONTROUTE 0x4 // bypass routing, use direct interface. The flag MSG_OOB is used to send 'out-of-band' data on sockets that support this notion (e.g. SOCK_STREAM); the underlying protocol must also support 'out-of-band' data. MSG_DONTROUTE is usually used only by diagnostic or routing programs."

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **flags** value of the I/O send flags for the specified SSL session (ssl).

See:

- wolfSSL_CTX_SetIORecv
- wolfSSL_CTX_SetIOSend
- wolfSSL_SetIOReadCtx

Return: none No returns.

Example

```
WOLFSSL* ssl = 0;
...
// Manually setting send flags to 0
wolfSSL_SetIOWriteFlags(ssl, 0);
...
```

C.69.2.14 function wolfSSL_SetIO_NetX

```
void wolfSSL_SetIO_NetX(
    WOLFSSL * ssl,
    NX_TCP_SOCKET * nxsocket,
    ULONG waitoption
)
```

This function sets the nxTcpSocket and nxWait members of the nxCtx struct within the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **nxsocket** a pointer to type NX_TCP_SOCKET that is set to the nxTcpSocket member of the nxCtx structure.
- **waitoption** a ULONG type that is set to the nxWait member of the nxCtx structure. With NX_NO_WAIT, or any wait option that can expire, the callbacks report WANT_READ/WANT_WRITE instead of a fatal error, so the wolfSSL call can be retried.

See:

- set_fd
- NetX_Send
- NetX_Receive

Return: none No returns.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
NX_TCP_SOCKET* nxsocket;
ULONG waitoption;
…
if(ssl != NULL || nxsocket != NULL || waitoption <= 0){
wolfSSL_SetIO_NetX(ssl, nxsocket, waitoption);
} else {
    // You need to pass in good parameters.
}
```

C.69.2.15 function wolfSSL_SetIO_NetX_Dtls

```
void wolfSSL_SetIO_NetX_Dtls(
    WOLFSSL * ssl,
```

```
    NX_UDP_SOCKET * nxsocket,
    NXD_ADDRESS nxdip,
    USHORT nxport,
    ULONG waitoption
)
```

This function configures the NetX Duo UDP context for a DTLS session. It stores the UDP socket, destination IP address (by value), destination port, and wait option into the WOLFSSL nxCtx structure. Requires WOLFSSL_NETX_DUO to be defined (ThreadX NetX Duo SDK).

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **nxsocket** a pointer to an NX_UDP_SOCKET already created and bound.
- **nxdip** the destination NXD_ADDRESS (passed by value; IPv4 or IPv6).
- **nxport** the destination UDP port number.
- **waitoption** a ULONG NetX wait option (e.g. NX_WAIT_FOREVER or ticks). With NX_NO_WAIT, or any wait option that can expire, NetX_SendTo reports WANT_WRITE instead of a fatal error. A receive that expires reports WANT_READ when the session is set non blocking, and a timeout otherwise so that DTLS can retransmit.

See:

- wolfSSL_SetIO_NetX
- NetX_SendTo
- NetX_ReceiveFrom

Return: none No returns.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
NX_UDP_SOCKET udpsocket;
NXD_ADDRESS   peeraddr;
USHORT        peerport = 4433;
ULONG         wait     = NX_WAIT_FOREVER;
// … initialise udpsocket and peeraddr …
wolfSSL_SetIO_NetX_Dtls(ssl, &udpsocket, peeraddr, peerport, wait);
```

C.69.2.16 function wolfSSL_CTX_SetGenCookie

```
void wolfSSL_CTX_SetGenCookie(
    WOLFSSL_CTX * ctx,
    CallbackGenCookie cb
)
```

This function sets the callback for the CBIOCookie member of the WOLFSSL_CTX structure. The CallbackGenCookie type is a function pointer and has the signature: int (*CallbackGenCookie)(WOLFSSL* ssl, unsigned char* buf, int sz, void* ctx);.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **cb** a CallbackGenCookie type function pointer with the signature of CallbackGenCookie.

See: CallbackGenCookie

Return: none No returns.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
…
int SetGenCookieCB(WOLFSSL* ssl, unsigned char* buf, int sz, void* ctx){
// Callback function body.
}
…
wolfSSL_CTX_SetGenCookie(ssl->ctx, SetGenCookieCB);
```

C.69.2.17 function wolfSSL_GetCookieCtx

```
void * wolfSSL_GetCookieCtx(
    WOLFSSL * ssl
)
```

This function returns the IOCB_CookieCtx member of the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **ssl** SSL object

See:

- wolfSSL_SetCookieCtx
- wolfSSL_CTX_SetGenCookie
- wolfSSL_SetCookieCtx

Return:

- pointer The function returns a void pointer value stored in the IOCB_CookieCtx.
- NULL if the WOLFSSL struct is NULL
- Cookie context pointer

Gets cookie context for DTLS.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
void* cookie;
...
cookie = wolfSSL_GetCookieCtx(ssl);
if(cookie != NULL){
// You have the cookie
}
```

Example

```
WOLFSSL* ssl;
void* ctx = wolfSSL_GetCookieCtx(ssl);
```

C.69.2.18 function wolfSSL_SetIO_ISOTP

```
int wolfSSL_SetIO_ISOTP(
    WOLFSSL * ssl,
    isotp_wolfssl_ctx * ctx,
    can_recv_fn recv_fn,
    can_send_fn send_fn,
    can_delay_fn delay_fn,
```

```
    word32 receive_delay,
    char * receive_buffer,
    int receive_buffer_size,
    void * arg
)
```

This function sets up the ISO-TP context if wolfSSL, for use when wolfSSL is compiled with WOLFSSL_ISOTP.

Parameters:

- **ssl** the wolfSSL context
- **ctx** a user created ISOTP context which this function initializes
- **recv_fn** a user CAN bus receive callback, returning the frame length, 0 if none arrived within timeout (ms), or negative on error
- **send_fn** a user CAN bus send callback
- **delay_fn** a user microsecond granularity delay function
- **receive_delay** a set amount of microseconds to delay each CAN bus packet
- **receive_buffer** a user supplied buffer to receive data, recommended that is allocated to ISOTP_DEFAULT_BUFFER_SIZE bytes
- **receive_buffer_size** - The size of receive_buffer
- **arg** an arbitrary pointer sent to recv_fn and send_fn

Return: 0 on success, WOLFSSL_CBIO_ERR_GENERAL on failure

Example

```
struct can_info can_con_info;
isotp_wolfssl_ctx isotp_ctx;
char *receive_buffer = malloc(ISOTP_DEFAULT_BUFFER_SIZE);
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(method);
WOLFSSL* ssl = wolfSSL_new(ctx);
...
wolfSSL_SetIO_ISOTP(ssl, &isotp_ctx, can_receive, can_send, can_delay, 0,
        receive_buffer, ISOTP_DEFAULT_BUFFER_SIZE, &can_con_info);
```

C.69.2.19 function wolfSSL_SSLDisableRead

```
void wolfSSL_SSLDisableRead(
    WOLFSSL * ssl
)
```

This function disables reading from the IO layer.

Parameters:

- **ssl** the wolfSSL context

See:

- wolfSSL_CTX_SetIORecv
- wolfSSL_SSLSetIORecv
- wolfSSL_SSLEnableRead

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(method);
WOLFSSL* ssl = wolfSSL_new(ctx);
wolfSSL_SSLDisableRead(ssl);
```

C.69.2.20 function wolfSSL_SSLEnableRead

```
void wolfSSL_SSLEnableRead(
    WOLFSSL * ssl
)
```

This function enables reading from the IO layer. Reading is enabled by default and should be used to undo wolfSSL_SSLDisableRead();.

Parameters:

- **ssl** the wolfSSL context

See:

- wolfSSL_CTX_SetIORecv
- wolfSSL_SSLSetIORecv
- wolfSSL_SSLEnableRead

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(method);
WOLFSSL* ssl = wolfSSL_new(ctx);
wolfSSL_SSLDisableRead(ssl);
...
wolfSSL_SSLEnableRead(ssl);
```

C.69.2.21 function wolfSSL_SetRecvFrom

```
WOLFSSL_API void wolfSSL_SetRecvFrom(
    WOLFSSL * ssl,
    WolfSSLRecvFrom recvFrom
)
```

Set a custom DTLS recvfrom callback for a WOLFSSL session.

Parameters:

- **ssl** A pointer to a WOLFSSL structure, created using wolfSSL_new().
- **recvFrom** The custom callback function to use for DTLS datagram receive.

See:

- WolfSSLRecvFrom
- wolfSSL_SetSendTo
- EmbedReceiveFrom
- wolfSSL_CTX_SetIORecv
- wolfSSL_SSLSetIORecv

This function allows you to specify a custom callback function for receiving datagrams (DTLS) using the `recvfrom`-style interface. The callback must match the WolfSSLRecvFrom function pointer type and is expected to behave like the POSIX `recvfrom()` function, including its return values and error handling.

Example

```
wolfSSL_SetRecvFrom(ssl, my_recvfrom_cb);
```

C.69.2.22 function wolfSSL_SetSendTo

```
WOLFSSL_API void wolfSSL_SetSendTo(
    WOLFSSL * ssl,
```

```
    WolfSSLSento sendTo
)
```

Set a custom DTLS sendto callback for a WOLFSSL session.

Parameters:

- **ssl** A pointer to a WOLFSSL structure, created using wolfSSL_new().
- **sendTo** The custom callback function to use for DTLS datagram send.

See:

- WolfSSLSento
- wolfSSL_SetRecvFrom
- EmbedSendTo
- wolfSSL_CTX_SetIOSend
- wolfSSL_SSLSetIOSend

This function allows you to specify a custom callback function for sending datagrams (DTLS) using the `sendto`-style interface. The callback must match the WolfSSLSento function pointer type and is expected to behave like the POSIX `sendto()` function, including its return values and error handling.

Example

```
wolfSSL_SetSendTo(ssl, my_sendto_cb);
```

C.69.2.23 function wolfIO_Select

```
int wolfIO_Select(
    SOCKET_T sockfd,
    int to_sec
)
```

Waits for socket to be ready for I/O with timeout.

Parameters:

- **sockfd** Socket file descriptor
- **to_sec** Timeout in seconds

See: wolfIO_TcpConnect

Return:

- 0 on success
- negative on error

Example

```
SOCKET_T sockfd;
int ret = wolfIO_Select(sockfd, 5);
```

C.69.2.24 function wolfIO_TcpConnect

```
int wolfIO_TcpConnect(
    SOCKET_T * sockfd,
    const char * ip,
    unsigned short port,
    int to_sec
)
```

Connects to TCP server with timeout.

Parameters:

- **sockfd** Pointer to socket file descriptor
- **ip** IP address string
- **port** Port number
- **to_sec** Timeout in seconds

See: wolfIO_TcpBind

Return:

- 0 on success
- negative on error

Example

```
SOCKET_T sockfd;
int ret = wolfIO_TcpConnect(&sockfd, "127.0.0.1", 443, 5);
```

C.69.2.25 function wolfIO_TcpAccept

```
int wolfIO_TcpAccept(
    SOCKET_T sockfd,
    SOCKADDR * peer_addr,
    XSOCKLENT * peer_len
)
```

Accepts TCP connection.

Parameters:

- **sockfd** Socket file descriptor
- **peer_addr** Peer address structure
- **peer_len** Peer address length

See: wolfIO_TcpBind

Return:

- Socket descriptor on success
- negative on error

Example

```
SOCKET_T sockfd;
SOCKADDR peer;
XSOCKLENT len = sizeof(peer);
int ret = wolfIO_TcpAccept(sockfd, &peer, &len);
```

C.69.2.26 function wolfIO_TcpBind

```
int wolfIO_TcpBind(
    SOCKET_T * sockfd,
    word16 port
)
```

Binds TCP socket to port.

Parameters:

- **sockfd** Pointer to socket file descriptor

- **port** Port number

See: wolfIO_TcpAccept

Return:

- 0 on success
- negative on error

Example

```
SOCKET_T sockfd;
int ret = wolfIO_TcpBind(&sockfd, 443);
```

C.69.2.27 function wolfIO_Send

```
int wolfIO_Send(
    SOCKET_T sd,
    char * buf,
    int sz,
    int wrFlags
)
```

Sends data on socket.

Parameters:

- **sd** Socket descriptor
- **buf** Buffer to send
- **sz** Buffer size
- **wrFlags** Write flags

See: wolfIO_Recv

Return:

- Number of bytes sent on success
- negative on error

Example

```
SOCKET_T sd;
char buf[100];
int ret = wolfIO_Send(sd, buf, sizeof(buf), 0);
```

C.69.2.28 function wolfIO_Recv

```
int wolfIO_Recv(
    SOCKET_T sd,
    char * buf,
    int sz,
    int rdFlags
)
```

Receives data from socket.

Parameters:

- **sd** Socket descriptor
- **buf** Buffer to receive into
- **sz** Buffer size
- **rdFlags** Read flags

See: wolfIO_Send

Return:

- Number of bytes received on success
- negative on error

Example

```
SOCKET_T sd;
char buf[100];
int ret = wolfIO_Recv(sd, buf, sizeof(buf), 0);
```

C.69.2.29 function wolfIO_SendTo

```
int wolfIO_SendTo(
    SOCKET_T sd,
    WOLFSSL_BIO_ADDR * addr,
    char * buf,
    int sz,
    int wrFlags
)
```

Sends datagram to address.

Parameters:

- **sd** Socket descriptor
- **addr** Destination address
- **buf** Buffer to send
- **sz** Buffer size
- **wrFlags** Write flags

See: wolfIO_RecvFrom

Return:

- Number of bytes sent on success
- negative on error

Example

```
SOCKET_T sd;
WOLFSSL_BIO_ADDR addr;
char buf[100];
int ret = wolfIO_SendTo(sd, &addr, buf, sizeof(buf), 0);
```

C.69.2.30 function wolfIO_RecvFrom

```
int wolfIO_RecvFrom(
    SOCKET_T sd,
    WOLFSSL_BIO_ADDR * addr,
    char * buf,
    int sz,
    int rdFlags
)
```

Receives datagram from address.

Parameters:

- **sd** Socket descriptor

- **addr** Source address
- **buf** Buffer to receive into
- **sz** Buffer size
- **rdFlags** Read flags

See: wolfIO_SendTo

Return:

- Number of bytes received on success
- negative on error

Example

```
SOCKET_T sd;
WOLFSSL_BIO_ADDR addr;
char buf[100];
int ret = wolfIO_RecvFrom(sd, &addr, buf, sizeof(buf), 0);
```

C.69.2.31 function wolfSSL_BioSend

```
int wolfSSL_BioSend(
    WOLFSSL * ssl,
    char * buf,
    int sz,
    void * ctx
)
```

BIO send callback.

Parameters:

- **ssl** SSL object
- **buf** Buffer to send
- **sz** Buffer size
- **ctx** Context pointer

See: wolfSSL_BioReceive

Return:

- Number of bytes sent on success
- negative on error

Example

```
WOLFSSL* ssl;
char buf[100];
int ret = wolfSSL_BioSend(ssl, buf, sizeof(buf), NULL);
```

C.69.2.32 function wolfSSL_BioReceive

```
int wolfSSL_BioReceive(
    WOLFSSL * ssl,
    char * buf,
    int sz,
    void * ctx
)
```

BIO receive callback.

Parameters:

- **ssl** SSL object
- **buf** Buffer to receive into
- **sz** Buffer size
- **ctx** Context pointer

See: wolfSSL_BioSend

Return:

- Number of bytes received on success
- negative on error

Example

```
WOLFSSL* ssl;
char buf[100];
int ret = wolfSSL_BioReceive(ssl, buf, sizeof(buf), NULL);
```

C.69.2.33 function EmbedReceiveFromMcast

```
int EmbedReceiveFromMcast(
    WOLFSSL * ssl,
    char * buf,
    int sz,
    void * ctx
)
```

Receives multicast datagram.

Parameters:

- **ssl** SSL object
- **buf** Buffer to receive into
- **sz** Buffer size
- **ctx** Context pointer

See: EmbedReceiveFrom

Return:

- Number of bytes received on success
- negative on error

Example

```
WOLFSSL* ssl;
char buf[100];
int ret = EmbedReceiveFromMcast(ssl, buf, sizeof(buf), NULL);
```

C.69.2.34 function wolfIO_HttpBuildRequestOcsp

```
int wolfIO_HttpBuildRequestOcsp(
    const char * domainName,
    const char * path,
    int ocspReqSz,
    unsigned char * buf,
    int bufSize
)
```

Builds HTTP OCSP request.

Parameters:

- **domainName** Domain name
- **path** URL path
- **ocspReqSz** OCSP request size
- **buf** Output buffer
- **bufSize** Buffer size

See: wolfIO_HttpProcessResponseOcsp

Return:

- Request size on success
- negative on error

Example

```
char buf[1024];
int ret = wolfIO_HttpBuildRequestOcsp("example.com", "/ocsp", 100,
                                      (unsigned char*)buf, sizeof(buf));
```

C.69.2.35 function wolfIO_HttpProcessResponseOcspGenericIO

```
int wolfIO_HttpProcessResponseOcspGenericIO(
    WolfSSLGenericIORecvCb ioCb,
    void * ioCbCtx,
    unsigned char ** respBuf,
    unsigned char * httpBuf,
    int httpBufSz,
    void * heap
)
```

Processes HTTP OCSP response with generic I/O.

Parameters:

- **ioCb** I/O callback
- **ioCbCtx** I/O callback context
- **respBuf** Response buffer pointer
- **httpBuf** HTTP buffer
- **httpBufSz** HTTP buffer size
- **heap** Heap hint

See: wolfIO_HttpProcessResponseOcsp

Return:

- 0 on success
- negative on error

Example

```
unsigned char* resp = NULL;
unsigned char httpBuf[1024];
int ret = wolfIO_HttpProcessResponseOcspGenericIO(myIoCb, ctx, &resp,
                                                  httpBuf,
                                                  sizeof(httpBuf), NULL);
```

C.69.2.36 function wolfIO_HttpProcessResponseOcsp

```
int wolfIO_HttpProcessResponseOcsp(
    int sfd,
    unsigned char ** respBuf,
```

```
    unsigned char * httpBuf,
    int httpBufSz,
    void * heap
)
```

Processes HTTP OCSP response.

Parameters:

- **sfd** Socket file descriptor
- **respBuf** Response buffer pointer
- **httpBuf** HTTP buffer
- **httpBufSz** HTTP buffer size
- **heap** Heap hint

See: wolfIO_HttpBuildRequestOcsp

Return:

- 0 on success
- negative on error

Example

```
int sfd;
unsigned char* resp = NULL;
unsigned char httpBuf[1024];
int ret = wolfIO_HttpProcessResponseOcsp(sfd, &resp, httpBuf,
                                         sizeof(httpBuf), NULL);
```

C.69.2.37 function EmbedOcspLookup

```
int EmbedOcspLookup(
    void * ctx,
    const char * url,
    int urlSz,
    byte * ocspReqBuf,
    int ocspReqSz,
    byte ** ocspRespBuf
)
```

OCSP lookup callback.

Parameters:

- **ctx** Context pointer
- **url** URL string
- **urlSz** URL size
- **ocspReqBuf** OCSP request buffer
- **ocspReqSz** OCSP request size
- **ocspRespBuf** OCSP response buffer pointer

See: EmbedOcspRespFree

Return:

- 0 on success
- negative on error

Example

```
byte* resp = NULL;
byte req[100];
int ret = EmbedOcspLookup(NULL, "http://example.com/ocsp", 25, req,
                          sizeof(req), &resp);
```

C.69.2.38 function wolfIO_HttpBuildRequestCrl

```
int wolfIO_HttpBuildRequestCrl(
    const char * url,
    int urlSz,
    const char * domainName,
    unsigned char * buf,
    int bufSize
)
```

Builds HTTP CRL request.

Parameters:

- **url** URL string
- **urlSz** URL size
- **domainName** Domain name
- **buf** Output buffer
- **bufSize** Buffer size

See: wolfIO_HttpProcessResponseCrl

Return:

- Request size on success
- negative on error

Example

```
char buf[1024];
int ret = wolfIO_HttpBuildRequestCrl("http://example.com/crl", 22,
                                     "example.com",
                                     (unsigned char*)buf, sizeof(buf));
```

C.69.2.39 function wolfIO_HttpProcessResponseCrl

```
int wolfIO_HttpProcessResponseCrl(
    WOLFSSL_CRL * crl,
    int sfd,
    unsigned char * httpBuf,
    int httpBufSz
)
```

Processes HTTP CRL response.

Parameters:

- **crl** CRL object
- **sfd** Socket file descriptor
- **httpBuf** HTTP buffer
- **httpBufSz** HTTP buffer size

See: wolfIO_HttpBuildRequestCrl

Return:

- 0 on success
- negative on error

Example

```
WOLFSSL_CRL crl;
int sfd;
unsigned char httpBuf[1024];
int ret = wolfIO_HttpProcessResponseCrl(&crl, sfd, httpBuf,
                                        sizeof(httpBuf));
```

C.69.2.40 function EmbedCrlLookup

```
int EmbedCrlLookup(
    WOLFSSL_CRL * crl,
    const char * url,
    int urlSz
)
```

CRL lookup callback.

Parameters:

- **crl** CRL object
- **url** URL string
- **urlSz** URL size

See: wolfIO_HttpBuildRequestCrl

Return:

- 0 on success
- negative on error

Example

```
WOLFSSL_CRL crl;
int ret = EmbedCrlLookup(&crl, "http://example.com/crl", 22);
```

C.69.2.41 function wolfIO_DecodeUrl

```
int wolfIO_DecodeUrl(
    const char * url,
    int urlSz,
    char * outName,
    char * outPath,
    unsigned short * outPort
)
```

Decodes URL into components.

Parameters:

- **url** URL string
- **urlSz** URL size
- **outName** Output domain name
- **outPath** Output path
- **outPort** Output port

See: wolfIO_HttpBuildRequest

Return:

- 0 on success
- negative on error

Example

```
char name[256], path[256];
unsigned short port;
int ret = wolfIO_DecodeUrl("http://example.com:443/path", 28, name,
                           path, &port);
```

C.69.2.42 function wolfIO_HttpBuildRequest

```
int wolfIO_HttpBuildRequest(
    const char * reqType,
    const char * domainName,
    const char * path,
    int pathLen,
    int reqSz,
    const char * contentType,
    unsigned char * buf,
    int bufSize
)
```

Builds generic HTTP request.

Parameters:

- **reqType** Request type (GET, POST, etc.)
- **domainName** Domain name
- **path** URL path
- **pathLen** Path length
- **reqSz** Request body size
- **contentType** Content type
- **buf** Output buffer
- **bufSize** Buffer size

See: wolfIO_HttpProcessResponse

Return:

- Request size on success
- negative on error

Example

```
char buf[1024];
int ret = wolfIO_HttpBuildRequest("POST", "example.com", "/api", 4,
                                  100, "application/json",
                                  (unsigned char*)buf, sizeof(buf));
```

C.69.2.43 function wolfIO_HttpProcessResponseGenericIO

```
int wolfIO_HttpProcessResponseGenericIO(
    WolfSSLGenericIORecvCb ioCb,
    void * ioCbCtx,
    const char ** appStrList,
    unsigned char ** respBuf,
    unsigned char * httpBuf,
    int httpBufSz,
```

```
    int dynType,
    void * heap
)
```

Processes HTTP response with generic I/O.

Parameters:

- **ioCb** I/O callback
- **ioCbCtx** I/O callback context
- **appStrList** Application string list
- **respBuf** Response buffer pointer
- **httpBuf** HTTP buffer
- **httpBufSz** HTTP buffer size
- **dynType** Dynamic type
- **heap** Heap hint

See: wolfIO_HttpProcessResponse

Return:

- 0 on success
- negative on error

Example

```
unsigned char* resp = NULL;
unsigned char httpBuf[1024];
const char* appStrs[] = {"200 OK", NULL};
int ret = wolfIO_HttpProcessResponseGenericIO(myIoCb, ctx, appStrs,
                                              &resp, httpBuf,
                                              sizeof(httpBuf), 0, NULL);
```

C.69.2.44 function wolfIO_HttpProcessResponse

```
int wolfIO_HttpProcessResponse(
    int sfd,
    const char ** appStrList,
    unsigned char ** respBuf,
    unsigned char * httpBuf,
    int httpBufSz,
    int dynType,
    void * heap
)
```

Processes HTTP response.

Parameters:

- **sfd** Socket file descriptor
- **appStrList** Application string list
- **respBuf** Response buffer pointer
- **httpBuf** HTTP buffer
- **httpBufSz** HTTP buffer size
- **dynType** Dynamic type
- **heap** Heap hint

See: wolfIO_HttpBuildRequest

Return:

- 0 on success
- negative on error

Example

```
int sfd;
unsigned char* resp = NULL;
unsigned char httpBuf[1024];
const char* appStrs[] = {"200 OK", NULL};
int ret = wolfIO_HttpProcessResponse(sfd, appStrs, &resp, httpBuf,
                                     sizeof(httpBuf), 0, NULL);
```

C.69.2.45 function wolfSSL_CTX_SetIOSend

```
void wolfSSL_CTX_SetIOSend(
    WOLFSSL_CTX * ctx,
    CallbackIOSend CBIOSend
)
```

Sets I/O send callback for context.

Parameters:

- **ctx** SSL context
- **CBIOSend** Send callback

See: wolfSSL_SSLSetIOSend

Return: none No returns

Example

```
WOLFSSL_CTX* ctx;
wolfSSL_CTX_SetIOSend(ctx, mySendCallback);
```

C.69.2.46 function wolfSSL_SSLSetIORecv

```
void wolfSSL_SSLSetIORecv(
    WOLFSSL * ssl,
    CallbackIORecv CBIORecv
)
```

Sets I/O receive callback for SSL object.

Parameters:

- **ssl** SSL object
- **CBIORecv** Receive callback

See: wolfSSL_CTX_SetIORecv

Return: none No returns

Example

```
WOLFSSL* ssl;
wolfSSL_SSLSetIORecv(ssl, myRecvCallback);
```

C.69.2.47 function wolfSSL_SSLSetIOSend

```
void wolfSSL_SSLSetIOSend(
    WOLFSSL * ssl,
    CallbackIOSend CBIOSend
)
```

Sets I/O send callback for SSL object.

Parameters:

- **ssl** SSL object
- **CBIOSend** Send callback

See: wolfSSL_CTX_SetIOSend

Return: none No returns

Example

```
WOLFSSL* ssl;
wolfSSL_SSLSetIOSend(ssl, mySendCallback);
```

C.69.2.48 function wolfSSL_SetIO_Mynewt

```
void wolfSSL_SetIO_Mynewt(
    WOLFSSL * ssl,
    struct mn_socket * mnSocket,
    struct mn_sockaddr_in * mnSockAddrIn
)
```

Sets I/O for Mynewt platform.

Parameters:

- **ssl** SSL object
- **mnSocket** Mynewt socket
- **mnSockAddrIn** Mynewt socket address

See: wolfSSL_SetIO_LwIP

Return: none No returns

Example

```
WOLFSSL* ssl;
struct mn_socket sock;
struct mn_sockaddr_in addr;
wolfSSL_SetIO_Mynewt(ssl, &sock, &addr);
```

C.69.2.49 function wolfSSL_SetIO_LwIP

```
int wolfSSL_SetIO_LwIP(
    WOLFSSL * ssl,
    void * pcb,
    tcp_recv_fn recv,
    tcp_sent_fn sent,
    void * arg
)
```

Sets I/O for LwIP platform.

Parameters:

- **ssl** SSL object
- **pcb** Protocol control block
- **recv** Receive callback
- **sent** Sent callback
- **arg** Argument pointer

See: wolfSSL_SetIO_Mynewt

Return:

- 0 on success
- negative on error

Example

```
WOLFSSL* ssl;
struct tcp_pcb* pcb;
int ret = wolfSSL_SetIO_LwIP(ssl, pcb, myRecv, mySent, NULL);
```

C.69.2.50 function wolfSSL_SetCookieCtx

```
void wolfSSL_SetCookieCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

Sets cookie context for DTLS.

Parameters:

- **ssl** SSL object
- **ctx** Cookie context

See: wolfSSL_GetCookieCtx

Return: none No returns

Example

```
WOLFSSL* ssl;
void* ctx;
wolfSSL_SetCookieCtx(ssl, ctx);
```

C.69.2.51 function wolfSSL_CTX_SetIOGetPeer

```
void wolfSSL_CTX_SetIOGetPeer(
    WOLFSSL_CTX * ctx,
    CallbackGetPeer cb
)
```

Sets get peer callback for context.

Parameters:

- **ctx** SSL context
- **cb** Get peer callback

See: wolfSSL_CTX_SetIOSetPeer

Return: none No returns

Example

```
WOLFSSL_CTX* ctx;
wolfSSL_CTX_SetIOGetPeer(ctx, myGetPeerCallback);
```

C.69.2.52 function wolfSSL_CTX_SetIOSetPeer

```
void wolfSSL_CTX_SetIOSetPeer(
    WOLFSSL_CTX * ctx,
    CallbackSetPeer cb
)
```

Sets set peer callback for context.

Parameters:

- **ctx** SSL context
- **cb** Set peer callback

See: wolfSSL_CTX_SetIOGetPeer

Return: none No returns

Example

```
WOLFSSL_CTX* ctx;
wolfSSL_CTX_SetIOSetPeer(ctx, mySetPeerCallback);
```

C.69.2.53 function EmbedGetPeer

```
int EmbedGetPeer(
    WOLFSSL * ssl,
    char * ip,
    int * ipSz,
    unsigned short * port,
    int * fam
)
```

Gets peer information.

Parameters:

- **ssl** SSL object
- **ip** IP address buffer
- **ipSz** IP address buffer size pointer
- **port** Port number pointer
- **fam** Address family pointer

See: EmbedSetPeer

Return:

- 0 on success
- negative on error

Example

```
WOLFSSL* ssl;
char ip[46];
int ipSz = sizeof(ip);
unsigned short port;
int fam;
int ret = EmbedGetPeer(ssl, ip, &ipSz, &port, &fam);
```

C.69.2.54 function EmbedSetPeer

```
int EmbedSetPeer(
    WOLFSSL * ssl,
    char * ip,
    int ipSz,
    unsigned short port,
    int fam
)
```

Sets peer information.

Parameters:

- **ssl** SSL object
- **ip** IP address string
- **ipSz** IP address string size
- **port** Port number
- **fam** Address family

See: EmbedGetPeer

Return:

- 0 on success
- negative on error

Example

```
WOLFSSL* ssl;
int ret = EmbedSetPeer(ssl, "127.0.0.1", 9, 443, AF_INET);
```

C.69.3 Source code

```
int EmbedReceive(WOLFSSL* ssl, char* buf, int sz, void* ctx);

int EmbedSend(WOLFSSL* ssl, char* buf, int sz, void* ctx);

int EmbedReceiveFrom(WOLFSSL* ssl, char* buf, int sz, void* ctx);

int EmbedSendTo(WOLFSSL* ssl, char* buf, int sz, void* ctx);

int EmbedGenerateCookie(WOLFSSL* ssl, byte* buf,
                                    int sz, void* ctx);

void EmbedOcspRespFree(void* ctx, byte* resp);

void wolfSSL_CTX_SetIORecv(WOLFSSL_CTX* ctx, CallbackIORecv CBIORecv);

void wolfSSL_SetIOReadCtx( WOLFSSL* ssl, void *ctx);

void wolfSSL_SetIOWriteCtx(WOLFSSL* ssl, void *ctx);

void* wolfSSL_GetIOReadCtx( WOLFSSL* ssl);

void* wolfSSL_GetIOWriteCtx(WOLFSSL* ssl);
```

```
void wolfSSL_SetIOReadFlags( WOLFSSL* ssl, int flags);

void wolfSSL_SetIOWriteFlags(WOLFSSL* ssl, int flags);

void wolfSSL_SetIO_NetX(WOLFSSL* ssl, NX_TCP_SOCKET* nxsocket,
                                      ULONG waitoption);

void wolfSSL_SetIO_NetX_Dtls(WOLFSSL* ssl, NX_UDP_SOCKET* nxsocket,
                             NXD_ADDRESS nxdip, USHORT nxport,
                             ULONG waitoption);

void  wolfSSL_CTX_SetGenCookie(WOLFSSL_CTX* ctx, CallbackGenCookie cb);

void* wolfSSL_GetCookieCtx(WOLFSSL* ssl);


int wolfSSL_SetIO_ISOTP(WOLFSSL *ssl, isotp_wolfssl_ctx *ctx,
        can_recv_fn recv_fn, can_send_fn send_fn, can_delay_fn delay_fn,
        word32 receive_delay, char *receive_buffer, int receive_buffer_size,
        void *arg);

void wolfSSL_SSLDisableRead(WOLFSSL *ssl);

void wolfSSL_SSLEnableRead(WOLFSSL *ssl);

WOLFSSL_API void wolfSSL_SetRecvFrom(WOLFSSL* ssl, WolfSSLRecvFrom recvFrom);

WOLFSSL_API void wolfSSL_SetSendTo(WOLFSSL* ssl, WolfSSLSento sendTo);

int wolfIO_Select(SOCKET_T sockfd, int to_sec);

int wolfIO_TcpConnect(SOCKET_T* sockfd, const char* ip,
                      unsigned short port, int to_sec);

int wolfIO_TcpAccept(SOCKET_T sockfd, SOCKADDR* peer_addr,
                     XSOCKLENT* peer_len);

int wolfIO_TcpBind(SOCKET_T* sockfd, word16 port);

int wolfIO_Send(SOCKET_T sd, char *buf, int sz, int wrFlags);

int wolfIO_Recv(SOCKET_T sd, char *buf, int sz, int rdFlags);

int wolfIO_SendTo(SOCKET_T sd, WOLFSSL_BIO_ADDR *addr, char *buf, int sz,
                  int wrFlags);

int wolfIO_RecvFrom(SOCKET_T sd, WOLFSSL_BIO_ADDR *addr, char *buf, int sz,
                    int rdFlags);

int wolfSSL_BioSend(WOLFSSL* ssl, char *buf, int sz, void *ctx);

int wolfSSL_BioReceive(WOLFSSL* ssl, char* buf, int sz, void* ctx);

int EmbedReceiveFromMcast(WOLFSSL *ssl, char *buf, int sz, void *ctx);
```

```
int wolfIO_HttpBuildRequestOcsp(const char* domainName, const char* path,
                                int ocspReqSz, unsigned char* buf,
                                int bufSize);

int wolfIO_HttpProcessResponseOcspGenericIO(WolfSSLGenericIORecvCb ioCb,
                                            void* ioCbCtx,
                                            unsigned char** respBuf,
                                            unsigned char* httpBuf,
                                            int httpBufSz, void* heap);

int wolfIO_HttpProcessResponseOcsp(int sfd, unsigned char** respBuf,
                                   unsigned char* httpBuf, int httpBufSz,
                                   void* heap);

int EmbedOcspLookup(void* ctx, const char* url, int urlSz,
                    byte* ocspReqBuf, int ocspReqSz, byte** ocspRespBuf);

int wolfIO_HttpBuildRequestCrl(const char* url, int urlSz,
                               const char* domainName, unsigned char* buf,
                               int bufSize);

int wolfIO_HttpProcessResponseCrl(WOLFSSL_CRL* crl, int sfd,
                                  unsigned char* httpBuf, int httpBufSz);

int EmbedCrlLookup(WOLFSSL_CRL* crl, const char* url, int urlSz);

int wolfIO_DecodeUrl(const char* url, int urlSz, char* outName,
                     char* outPath, unsigned short* outPort);

int wolfIO_HttpBuildRequest(const char* reqType, const char* domainName,
                            const char* path, int pathLen, int reqSz,
                            const char* contentType, unsigned char* buf,
                            int bufSize);

int wolfIO_HttpProcessResponseGenericIO(WolfSSLGenericIORecvCb ioCb,
                                        void* ioCbCtx,
                                        const char** appStrList,
                                        unsigned char** respBuf,
                                        unsigned char* httpBuf,
                                        int httpBufSz, int dynType,
                                        void* heap);

int wolfIO_HttpProcessResponse(int sfd, const char** appStrList,
                               unsigned char** respBuf,
                               unsigned char* httpBuf, int httpBufSz,
                               int dynType, void* heap);

void wolfSSL_CTX_SetIOSend(WOLFSSL_CTX *ctx, CallbackIOSend CBIOSend);

void wolfSSL_SSLSetIORecv(WOLFSSL *ssl, CallbackIORecv CBIORecv);

void wolfSSL_SSLSetIOSend(WOLFSSL *ssl, CallbackIOSend CBIOSend);
```

```
void wolfSSL_SetIO_Mynewt(WOLFSSL* ssl, struct mn_socket* mnSocket,
                          struct mn_sockaddr_in* mnSockAddrIn);

int wolfSSL_SetIO_LwIP(WOLFSSL* ssl, void *pcb, tcp_recv_fn recv,
                       tcp_sent_fn sent, void *arg);

void wolfSSL_SetCookieCtx(WOLFSSL* ssl, void *ctx);

void* wolfSSL_GetCookieCtx(WOLFSSL* ssl);

void wolfSSL_CTX_SetIOGetPeer(WOLFSSL_CTX* ctx, CallbackGetPeer cb);

void wolfSSL_CTX_SetIOSetPeer(WOLFSSL_CTX* ctx, CallbackSetPeer cb);

int EmbedGetPeer(WOLFSSL* ssl, char* ip, int* ipSz, unsigned short* port,
                 int* fam);

int EmbedSetPeer(WOLFSSL* ssl, char* ip, int ipSz, unsigned short port,
                 int fam);
```

D SSL/TLS Overview

D.1 General Architecture

The wolfSSL (formerly CyaSSL) embedded SSL library implements SSL 3.0, TLS 1.0, TLS 1.1, TLS 1.2, and TLS 1.3 protocols. TLS 1.3 is currently the most secure and up to date version of the standard. wolfSSL does not support SSL 2.0 due to the fact that it has been insecure for several years.

The TLS protocol in wolfSSL is implemented as defined in RFC 5246 (https://tools.ietf.org/html/rfc5246). Two record layer protocols exist within SSL - the message layer and the handshake layer. Handshake messages are used to negotiate a common cipher suite, create secrets, and enable a secure connection. The message layer encapsulates the handshake layer while also supporting alert processing and application data transfer.

A general diagram of how the SSL protocol fits into existing protocols can be seen in **Figure 1**. SSL sits in between the Transport and Application layers of the OSI model, where any number of protocols (including TCP/IP, Bluetooth, etc.) may act as the transport medium. Application protocols are layered on top of SSL (such as HTTP, FTP, and SMTP).

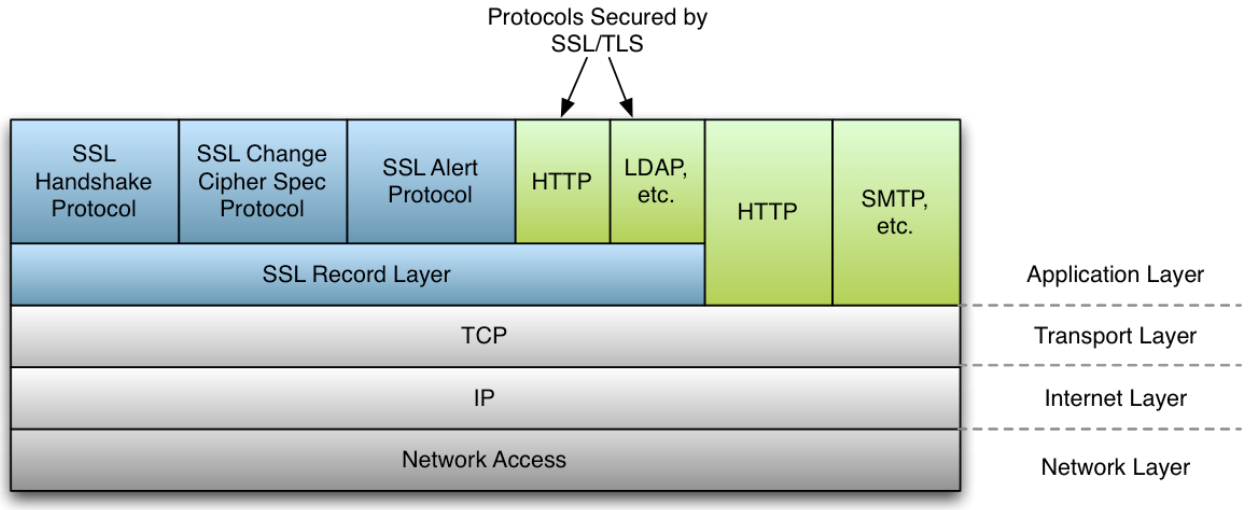

Figure 5: SSL Protocol Diagram

D.2 SSL Handshake

The SSL handshake involves several steps, some of which are optional depending on what options the SSL client and server have been configured with. Below, in **Figure 2**, you will find a simplified diagram of the SSL handshake process.

D.3 Differences between SSL and TLS Protocol Versions

SSL (Secure Sockets Layer) and TLS (Transport Security Layer) are both cryptographic protocols which provide secure communication over networks. These two protocols (and the several versions of each) are in widespread use today in applications ranging from web browsing to e-mail to instant messaging and VoIP. Each protocol, and the underlying versions of each, are slightly different from the other.

Below you will find both an explanation of, and the major differences between the different SSL and TLS protocol versions. For specific details about each protocol, please reference the RFC specification mentioned.

COPYRIGHT ©2024 wolfSSL Inc.

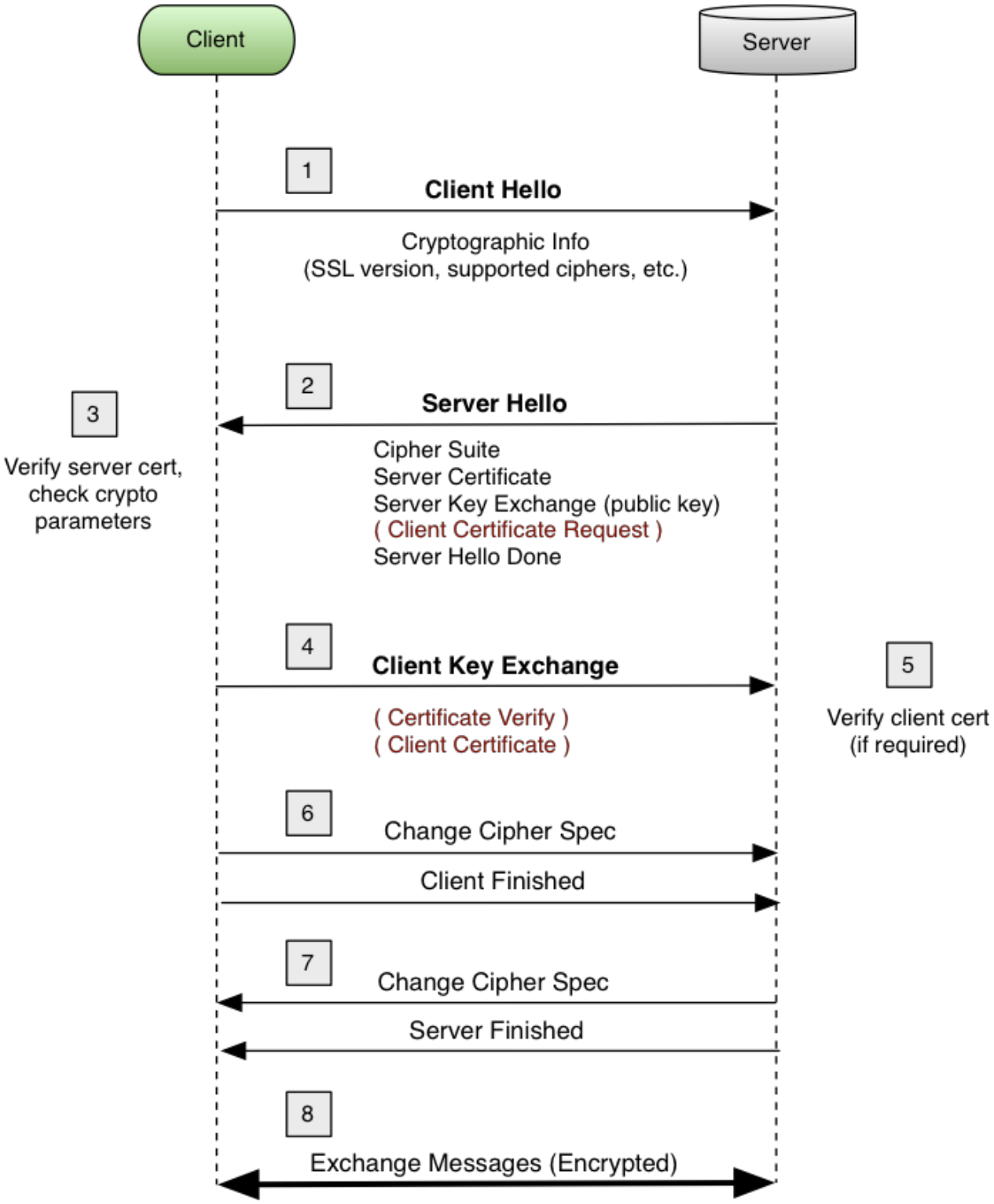

Figure 6: SSL Handshake Diagram

COPYRIGHT ©2024 wolfSSL Inc.

D.3.1 SSL 3.0

This protocol was released in 1996 but began with the creation of SSL 1.0 developed by Netscape. Version 1.0 wasn't released, and version 2.0 had a number of security flaws, thus leading to the release of SSL 3.0. Some major improvements of SSL 3.0 over SSL 2.0 are:

- Separation of the transport of data from the message layer
- Use of a full 128 bits of keying material even when using the Export cipher
- Ability of the client and server to send chains of certificates, thus allowing organizations to use certificate hierarchy which is more than two certificates deep.
- Implementing a generalized key exchange protocol, allowing Diffie-Hellman and Fortezza key exchanges as well as non-RSA certificates.
- Allowing for record compression and decompression
- Ability to fall back to SSL 2.0 when a 2.0 client is encountered

D.3.2 TLS 1.0

This protocol was first defined in RFC 2246 in January of 1999. This was an upgrade from SSL 3.0 and the differences were not dramatic, but they are significant enough that SSL 3.0 and TLS 1.0 don't interoperate. Some of the major differences between SSL 3.0 and TLS 1.0 are:

- Key derivation functions are different
- MACs are different - SSL 3.0 uses a modification of an early HMAC while TLS 1.0 uses HMAC.
- The Finished messages are different
- TLS has more alerts
- TLS requires DSS/DH support

D.3.3 TLS 1.1

This protocol was defined in RFC 4346 in April of 2006, and is an update to TLS 1.0. The major changes are:

- The Implicit Initialization Vector (IV) is replaced with an explicit IV to protect against Cipher block chaining (CBC) attacks.
- Handling of padded errors is changed to use the `bad_record_mac` alert rather than the `decryption_failed` alert to protect against CBC attacks.
- IANA registries are defined for protocol parameters
- Premature closes no longer cause a session to be non-resumable.

D.3.4 TLS 1.2

This protocol was defined in RFC 5246 in August of 2008. Based on TLS 1.1, TLS 1.2 contains improved flexibility. The major differences include:

- The MD5/SHA-1 combination in the pseudorandom function (PRF) was replaced with cipher-suite-specified PRFs.
- The MD5/SHA-1 combination in the digitally-signed element was replaced with a single hash. Signed elements include a field explicitly specifying the hash algorithm used.
- There was substantial cleanup to the client's and server's ability to specify which hash and signature algorithms they will accept.
- Addition of support for authenticated encryption with additional data modes.
- TLS Extensions definition and AES Cipher Suites were merged in.
- Tighter checking of EncryptedPreMasterSecret version numbers.
- Many of the requirements were tightened
- `Verify_data` length depends on the cipher suite
- Description of Bleichenbacher/Dlima attack defenses cleaned up.

COPYRIGHT ©2024 wolfSSL Inc.

D.3.5 TLS 1.3

This protocol was defined in RFC 8446 in August of 2018. TLS 1.3 contains improved security and speed. The major differences include:

- The list of supported symmetric algorithms has been pruned of all legacy algorithms. The remaining algorithms all use Authenticated Encryption with Associated Data (AEAD) algorithms.
- A zero-RTT (0-RTT) mode was added, saving a round-trip at connection setup for some application data at the cost of certain security properties.
- All handshake messages after the ServerHello are now encrypted.
- Key derivation functions have been re-designed, with the HMAC-based Extract-and-Expand Key Derivation Function (HKDF) being used as a primitive.
- The handshake state machine has been restructured to be more consistent and remove superfluous messages.
- ECC is now in the base spec and includes new signature algorithms. Point format negotiation has been removed in favor of single point format for each curve.
- Compression, custom DHE groups, and DSA have been removed, RSA padding now uses PSS.
- TLS 1.2 version negotiation verification mechanism was deprecated in favor of a version list in an extension.
- Session resumption with and without server-side state and the PSK-based ciphersuites of earlier versions of TLS have been replaced by a single new PSK exchange.

COPYRIGHT ©2024 wolfSSL Inc.

E RFCs, Specifications, and Reference

E.1 Protocols

- SSL v3.0 - IETF Draft
- TLS v1.0 - RFC2246
- TLS v1.1 - RFC4346
- TLS v1.2 - RFC5246
- TLS v1.3 - RFC8446
- DTLS - RFC4347 Specification document
- IPv4 - Wikipedia
- IPv6 - Wikipedia

E.2 Stream Ciphers

- Stream Cipher Information - Wikipedia
- RC4 / ARC4 - IETF Draft Wikipedia

E.3 Block Ciphers

- Block Cipher Information - Wikipedia
- AES - NIST Publication Wikipedia
- AES-GCM - NIST Specification
- AES-NI - Intel Software Network
- DES/3DES - NIST Publication Wikipedia

E.4 Hashing Functions

- SHA - NIST FIPS180-1 Publication NIST FIPS180-2 Publication Wikipedia
- MD4 - RFC1320
- MD5 - RFC1321
- RIPEMD-160 - Specification document

E.5 Public Key Cryptography

- Diffie-Hellman - Wikipedia
- RSA - MIT Paper Wikipedia
- DSA/DSS - NIST FIPS186-3
- ECDSA - Specification Document
- X.509 - RFC3279
- ASN.1 - Specification Document Wikipedia
- PSK - RFC4279

E.6 Other

- PKCS#5, PBKDF1, PBKDF2 - RFC2898
- PKCS#8 - RFC5208
- PKCS#12 - Wikipedia

COPYRIGHT ©2024 wolfSSL Inc.

F Error Codes

F.1 wolfSSL Error Codes

wolfSSL (formerly CyaSSL) error codes can be found in `wolfssl/ssl.h`. For detailed descriptions of the following errors, see the OpenSSL man page for `SSL_get_error` (`man SSL_get_error`).

Error Code Enum	Error Code	Error Description
SSL_ERROR_WANT_READ	2	
SSL_ERROR_WANT_WRITE	3	
SSL_ERROR_WANT_CONNECT	7	
SSL_ERROR_WANT_ACCEPT	8	
SSL_ERROR_SYSCALL	5	
SSL_ERROR_WANT_X509_LOOKUP	83	
SSL_ERROR_ZERO_RETURN	6	
SSL_ERROR_SSL	85	

Additional wolfSSL error codes can be found in `wolfssl/error-ssl.h`

Error Code Enum	Error Code	Error Description
INPUT_CASE_ERROR	-301	process input state error
PREFIX_ERROR	-302	bad index to key rounds
MEMORY_ERROR	-303	out of memory
VERIFY_FINISHED_ERROR	-304	verify problem on finished
VERIFY_MAC_ERROR	-305	verify mac problem
PARSE_ERROR	-306	parse error on header
UNKNOWN_HANDSHAKE_TYPE	-307	weird handshake type
SOCKET_ERROR_E	-308	error state on socket
SOCKET_NODATA	-309	expected data, not there
INCOMPLETE_DATA	-310	don't have enough data to complete task
UNKNOWN_RECORD_TYPE	-311	unknown type in record hdr
DECRYPT_ERROR	-312	error during decryption
FATAL_ERROR	-313	revcd alert fatal error
ENCRYPT_ERROR	-314	error during encryption
FREAD_ERROR	-315	fread problem
NO_PEER_KEY	-316	need peer's key
NO_PRIVATE_KEY	-317	need the private key
RSA_PRIVATE_ERROR	-318	error during rsa priv op
NO_DH_PARAMS	-319	server missing DH params
BUILD_MSG_ERROR	-320	build message failure
BAD_HELLO	-321	client hello malformed
DOMAIN_NAME_MISMATCH	-322	peer subject name mismatch
WANT_READ	-323	want read, call again
NOT_READY_ERROR	-324	handshake layer not ready
VERSION_ERROR	-326	record layer version error
WANT_WRITE	-327	want write, call again
BUFFER_ERROR	-328	malformed buffer input
VERIFY_CERT_ERROR	-329	verify cert error
VERIFY_SIGN_ERROR	-330	verify sign error
CLIENT_ID_ERROR	-331	psk client identity error
SERVER_HINT_ERROR	-332	psk server hint error

COPYRIGHT ©2024 wolfSSL Inc.

Error Code Enum	Error Code	Error Description
PSK_KEY_ERROR	-333	psk key error
GETTIME_ERROR	-337	gettimeofday failed ???
GETITIMER_ERROR	-338	getitimer failed ???
SIGACT_ERROR	-339	sigaction failed ???
SETITIMER_ERROR	-340	setitimer failed ???
LENGTH_ERROR	-341	record layer length error
PEER_KEY_ERROR	-342	cant decode peer key
ZERO_RETURN	-343	peer sent close notify
SIDE_ERROR	-344	wrong client/server type
NO_PEER_CERT	-345	peer didn't send key
ECC_CURVETYPE_ERROR	-350	Bad ECC Curve Type
ECC_CURVE_ERROR	-351	Bad ECC Curve
ECC_PEERKEY_ERROR	-352	Bad Peer ECC Key
ECC_MAKEKEY_ERROR	-353	Bad Make ECC Key
ECC_EXPORT_ERROR	-354	Bad ECC Export Key
ECC_SHARED_ERROR	-355	Bad ECC Shared Secret
NOT_CA_ERROR	-357	Not CA cert error
BAD_CERT_MANAGER_ERROR	-359	Bad Cert Manager
OCSP_CERT_REVOKED	-360	OCSP Certificate revoked
CRL_CERT_REVOKED	-361	CRL Certificate revoked
CRL_MISSING	-362	CRL Not loaded
MONITOR_SETUP_E	-363	CRL Monitor setup error
THREAD_CREATE_E	-364	Thread Create Error
OCSP_NEED_URL	-365	OCSP need an URL for lookup
OCSP_CERT_UNKNOWN	-366	OCSP responder doesn't know
OCSP_LOOKUP_FAIL	-367	OCSP lookup not successful
MAX_CHAIN_ERROR	-368	max chain depth exceeded
COOKIE_ERROR	-369	dtls cookie error
SEQUENCE_ERROR	-370	dtls sequence error
SUITES_ERROR	-371	suites pointer error
OUT_OF_ORDER_E	-373	out of order message
BAD_KEA_TYPE_E	-374	bad KEA type found
SANITY_CIPHER_E	-375	sanity check on cipher error
RECV_OVERFLOW_E	-376	RXCB returned more than rqed
GEN_COOKIE_E	-377	Generate Cookie Error
NO_PEER_VERIFY	-378	Need peer cert verify Error
FWRITE_ERROR	-379	fwrite problem
CACHE_MATCH_ERROR	-380	cache hrd match error
UNKNOWN_SNI_HOST_NAME_E	-381	Unrecognized host name Error
UNKNOWN_MAX_FRAG_LEN_E	-382	Unrecognized max frag len Error
KEYUSE_SIGNATURE_E	-383	KeyUse digSignature error
KEYUSE_ENCIPHER_E	-385	KeyUse KeyEncipher error
EXTKEYUSE_AUTH_E	-386	ExtKeyUse server|client_auth
SEND_OOB_READ_E	-387	Send Cb out of bounds read
SECURE_RENEGOTIATION_E	-388	Invalid renegotiation info
SESSION_TICKET_LEN_E	-389	Session Ticket too large
SESSION_TICKET_EXPECT_E	-390	Session Ticket missing
SCR_DIFFERENT_CERT_E	-391	SCR Different cert error
SESSION_SECRET_CB_E	-392	Session secret CB fcn failure
NO_CHANGE_CIPHER_E	-393	Finished before change cipher
SANITY_MSG_E	-394	Sanity check on msg order error
DUPLICATE_MST_E	-395	Duplicate message error

Error Code Enum	Error Code	Error Description
SNI_UNSUPPORTED	-396	SSL 3.0 does not support SNI
SOCKET_PEER_CLOSED_E	-397	Underlying transport closed
BAD_TICKET_KEY_CB_SZ	-398	Bad session ticket key cb size
BAD_TICKET_MSG_SZ	-399	Bad session ticket msg size
BAD_TICKET_ENCRYPT	-400	Bad user ticket encrypt
DH_KEY_SIZE_E	-401	DH key too small
SNI_ABSENT_ERROR	-402	No SNI request
RSA_SIGN_FAULT	-403	RSA sign fault
HANDSHAKE_SIZE_ERROR	-404	Handshake message too large
UNKNOWN_ALPN_PROTOCOL_NAME_E	-405	Unrecognized protocol name error
BAD_CERTIFICATE_STATUS_ERROR	-406	Bad certificate status message
OCSP_INVALID_STATUS	-407	Invalid OCSP status
OCSP_WANT_READ	-408	OCSP callback response
RSA_KEY_SIZE_E	-409	RSA key too small
ECC_KEY_SIZE_E	-410	ECC key too small
DTLS_EXPORT_VER_E	-411	Export version error
INPUT_SIZE_E	-412	Input size too big error
CTX_INIT_MUTEX_E	-413	Initialize ctx mutex error
EXT_MASTER_SECRET_NEEDED_E	-414	Need EMS enabled to resume
DTLS_POOL_SZ_E	-415	Exceeded DTLS pool size
DECODE_E	-416	Decode handshake message error
HTTP_TIMEOUT	-417	HTTP timeout for OCSP or CRL req
WRITE_DUP_READ_E	-418	Write dup write side can't read
WRITE_DUP_WRITE_E	-419	Write dup read side can't write
INVALID_CERT_CTX_E	-420	TLS cert ctx not matching
BAD_KEY_SHARE_DATA	-421	Key share data invalid
MISSING_HANDSHAKE_DATA	-422	Handshake message missing data
BAD_BINDER	-423	Binder does not match
EXT_NOT_ALLOWED	-424	Extension not allowed in msg
INVALID_PARAMETER	-425	Security parameter invalid
MCAST_HIGHWATER_CB_E	-426	Multicast highwater cb err
ALERT_COUNT_E	-427	Alert count exceeded err
EXT_MISSING	-428	Required extension not found
UNSUPPORTED_EXTENSION	-429	TLSX not requested by client
PRF_MISSING	-430	PRF not compiled in
DTLS_RETX_OVER_TX	-431	Retransmit DTLS flight over
DH_PARAMS_NOT_FFDHE_E	-432	DH params from server not FFDHE
TCA_INVALID_ID_TYPE	-433	TLSX TCA ID type invalid
TCA_ABSENT_ERROR	-434	TLSX TCA ID no response

Negotiation Parameter Errors

Error Code Enum	Error Code	Error Description
UNSUPPORTED_SUITE	-500	Unsupported cipher suite
MATCH_SUITE_ERROR	-501	Can't match cipher suite
COMPRESSION_ERROR	-502	Compression mismatch
KEY_SHARE_ERROR	-503	Key share mismatch
POST_HAND_AUTH_ERROR	-504	Client won't do post-hand auth
HRR_COOKIE_ERROR	-505	HRR msg cookie mismatch

COPYRIGHT ©2024 wolfSSL Inc.

F.2 wolfCrypt Error Codes

wolfCrypt error codes can be found in `wolfssl/wolfcrypt/error.h`.

Error Code Enum	Error Code	Error Description
OPEN_RAN_E	-101	opening random device error
READ_RAN_E	-102	reading random device error
WINCRYPT_E	-103	windows crypt init error
CRYPTGEN_E	-104	windows crypt generation error
RAN_BLOCK_E	-105	reading random device would block
BAD_MUTEX_E	-106	Bad mutex operation
MP_INIT_E	-110	mp_init error state
MP_READ_E	-111	mp_read error state
MP_EXPTMOD_E	-112	mp_exptmod error state
MP_TO_E	-113	mp_to_xxx error state, can't convert
MP_SUB_E	-114	mp_sub error state, can't subtract
MP_ADD_E	-115	mp_add error state, can't add
MP_MUL_E	-116	mp_mul error state, can't multiply
MP_MULMOD_E	-117	mp_mulmod error state, can't multiply mod
MP_MOD_E	-118	mp_mod error state, can't mod
MP_INVMOD_E	-119	mp_invmod error state, can't inv mod
MP_CMP_E	-120	mp_cmp error state
MP_ZERO_E	-121	got a mp zero result, not expected
MEMORY_E	-125	out of memory error
RSA_WRONG_TYPE_E	-130	RSA wrong block type for RSA function
RSA_BUFFER_E	-131	RSA buffer error, output too small or input too large
BUFFER_E	-132	output buffer too small or input too large
ALGO_ID_E	-133	setting algo id error
PUBLIC_KEY_E	-134	setting public key error
DATE_E	-135	setting date validity error
SUBJECT_E	-136	setting subject name error
ISSUER_E	-137	setting issuer name error
CA_TRUE_E	-138	setting CA basic constraint true error
EXTENSIONS_E	-139	setting extensions error
ASN_PARSE_E	-140	ASN parsing error, invalid input
ASN_VERSION_E	-141	ASN version error, invalid number
ASN_GETINT_E	-142	ASN get big int error, invalid data
ASN_RSA_KEY_E	-143	ASN key init error, invalid input
ASN_OBJECT_ID_E	-144	ASN object id error, invalid id
ASN_TAG_NULL_E	-145	ASN tag error, not null
ASN_EXPECT_0_E	-146	ASN expect error, not zero
ASN_BITSTR_E	-147	ASN bit string error, wrong id
ASN_UNKNOWN_OID_E	-148	ASN oid error, unknown sum id
ASN_DATE_SZ_E	-149	ASN date error, bad size
ASN_BEFORE_DATE_E	-150	ASN date error, current date before
ASN_AFTER_DATE_E	-151	ASN date error, current date after
ASN_SIG_OID_E	-152	ASN signature error, mismatched oid
ASN_TIME_E	-153	ASN time error, unknown time type
ASN_INPUT_E	-154	ASN input error, not enough data
ASN_SIG_CONFIRM_E	-155	ASN sig error, confirm failure
ASN_SIG_HASH_E	-156	ASN sig error, unsupported hash type
ASN_SIG_KEY_E	-157	ASN sig error, unsupported key type
ASN_DH_KEY_E	-158	ASN key init error, invalid input
ASN_CRIT_EXT_E	-160	ASN unsupported critical extension

Error Code Enum	Error Code	Error Description
ECC_BAD_ARG_E	-170	ECC input argument of wrong type
ASN_ECC_KEY_E	-171	ASN ECC bad input
ECC_CURVE_OID_E	-172	Unsupported ECC OID curve type
BAD_FUNC_ARG	-173	Bad function argument provided
NOT_COMPILED_IN	-174	Feature not compiled in
UNICODE_SIZE_E	-175	Unicode password too big
NO_PASSWORD	-176	no password provided by user
ALT_NAME_E	-177	alt name size problem, too big
AES_GCM_AUTH_E	-180	AES-GCM Authentication check failure
AES_CCM_AUTH_E	-181	AES-CCM Authentication check failure
CAVIUM_INIT_E	-182	Cavium Init type error
COMPRESS_INIT_E	-183	Compress init error
COMPRESS_E	-184	Compress error
DECOMPRESS_INIT_E	-185	DeCompress init error
DECOMPRESS_E	-186	DeCompress error
BAD_ALIGN_E	-187	Bad alignment for operation, no alloc
ASN_NO_SIGNER_E	-188	ASN sig error, no CA signer to verify certificate
ASN_CRL_CONFIRM_E	-189	ASN CRL no signer to confirm failure
ASN_CRL_NO_SIGNER_E	-190	ASN CRL no signer to confirm failure
ASN_OCSP_CONFIRM_E	-191	ASN OCSP signature confirm failure
BAD_ENC_STATE_E	-192	Bad ecc enc state operation
BAD_PADDING_E	-193	Bad padding, msg not correct length
REQ_ATTRIBUTE_E	-194	Setting cert request attributes error
PKCS7_OID_E	-195	PKCS#7, mismatched OID error
PKCS7_RECIP_E	-196	PKCS#7, recipient error
FIPS_NOT_ALLOWED_E	-197	FIPS not allowed error
ASN_NAME_INVALID_E	-198	ASN name constraint error
RNG_FAILURE_E	-199	RNG Failed, Reinitialize
HMAC_MIN_KEYLEN_E	-200	FIPS Mode HMAC Minimum Key Length error
RSA_PAD_E	-201	RSA Padding Error
LENGTH_ONLY_E	-202	Returning output length only
IN_CORE_FIPS_E	-203	In Core Integrity check failure
AES_KAT_FIPS_E	-204	AES KAT failure
DES3_KAT_FIPS_E	-205	DES3 KAT failure
HMAC_KAT_FIPS_E	-206	HMAC KAT failure
RSA_KAT_FIPS_E	-207	RSA KAT failure
DRBG_KAT_FIPS_E	-208	HASH DRBG KAT failure
DRBG_CONT_FIPS_E	-209	HASH DRBG Continuous test failure
AESGCM_KAT_FIPS_E	-210	AESGCM KAT failure
THREAD_STORE_KEY_E	-211	Thread local storage key create failure
THREAD_STORE_SET_E	-212	Thread local storage key set failure
MAC_CMP_FAILED_E	-213	MAC comparison failed
IS_POINT_E	-214	ECC is point on curve failed
ECC_INF_E	-215	ECC point infinity error
ECC_PRIV_KEY_E	-216	ECC private key not valid error
SRP_CALL_ORDER_E	-217	SRP function called in the wrong order
SRP_VERIFY_E	-218	SRP proof verification failed
SRP_BAD_KEY_E	-219	SRP bad ephemeral values
ASN_NO_SKID	-220	ASN no Subject Key Identifier found
ASN_NO_AKID	-221	ASN no Authority Key Identifier found
ASN_NO_KEYUSAGE	-223	ASN no Key Usage found
SKID_E	-224	Setting Subject Key Identifier error

Error Code Enum	Error Code	Error Description
AKID_E	-225	Setting Authority Key Identifier error
KEYUSAGE_E	-226	Bad Key Usage value
CERTPOLICIES_E	-227	Setting Certificate Policies error
WC_INIT_E	-228	wolfCrypt failed to initialize
SIG_VERIFY_E	-229	wolfCrypt signature verify error
BAD_PKCS7_SIGNEEDS_CHECKCOND_E	-230	Bad condition variable operation
SIG_TYPE_E	-231	Signature Type not enabled/available
HASH_TYPE_E	-232	Hash Type not enabled/available
WC_KEY_SIZE_E	-234	Key size error, either too small or large
ASN_COUNTRY_SIZE_E	-235	ASN Cert Gen, invalid country code size
MISSING_RNG_E	-236	RNG required but not provided
ASN_PATHLEN_SIZE_E	-237	ASN CA path length too large error
ASN_PATHLEN_INV_E	-238	ASN CA path length inversion error
BAD_KEYWRAP_ALG_E	-239	Algorithm error with keywrap
BAD_KEYWRAP_IV_E	-240	Decrypted AES key wrap IV incorrect
WC_CLEANUP_E	-241	wolfCrypt cleanup failed
ECC_CDH_KAT_FIPS_E	-242	ECC CDH known answer test failure
DH_CHECK_PUB_E	-243	DH check public key error
BAD_PATH_ERROR	-244	Bad path for opendir
ASYNC_OP_E	-245	Async operation error
ECC_PRIVATEONLY_E	-246	Invalid use of private only ECC key
EXTKEYUSAGE_E	-247	Bad extended key usage value
WC_HW_E	-248	Error with hardware crypto use
WC_HW_WAIT_E	-249	Hardware waiting on resource
PSS_SALTLEN_E	-250	PSS length of salt is too long for hash
PRIME_GEN_E	-251	Failure finding a prime
BER_INDEF_E	-252	Cannot decode indefinite length BER
RSA_OUT_OF_RANGE_E	-253	Ciphertext to decrypt out of range
RSAPSS_PAT_FIPS_E	-254	RSA-PSS PAT failure
ECDSA_PAT_FIPS_E	-255	ECDSA PAT failure
DH_KAT_FIPS_E	-256	DH KAT failure
AESCCM_KAT_FIPS_E	-257	AESCCM KAT failure
SHA3_KAT_FIPS_E	-258	SHA-3 KAT failure
ECDHE_KAT_FIPS_E	-259	ECDHE KAT failure
AES_GCM_OVERFLOW_E	-260	AES-GCM invocation counter overflow
AES_CCM_OVERFLOW_E	-261	AES-CCM invocation counter overflow
RSA_KEY_PAIR_E	-262	RSA Key Pair-Wise consistency check fail
DH_CHECK_PRIVE_E	-263	DH check private key error
WC_AFALG_SOCK_E	-264	AF_ALG socket error
WC_DEVCRYPTO_E	-265	/dev/crypto error
ZLIB_INIT_ERROR	-266	Zlib init error
ZLIB_COMPRESS_ERROR	-267	Zlib compression error
ZLIB_DECOMPRESS_ERROR	-268	Zlib decompression error
PKCS7_NO_SIGNER_E	-269	No signer in PKCS7 signed data msg
WC_PKCS7_WANT_READ_E	-270	PKCS7 stream operation wants more input
CRYPTOCB_UNAVAILABLE	-271	Crypto callback unavailable
PKCS7_SIGNEEDS_CHECK	-272	Signature needs verified by caller
ASN_SELF_SIGNED_E	-275	ASN self-signed certificate error
MIN_CODE_E	-300	errors -101 - -299

F.3 Common Error Codes and their Solution

There are several error codes that commonly happen when getting an application up and running with wolfSSL.

F.3.1 ASN_NO_SIGNER_E (-188)

This error occurs when using a certificate and the signing CA certificate was not loaded. This can be seen using the wolfSSL example server or client against another client or server, for example connecting to Google using the wolfSSL example client:

```
./examples/client/client -g -h www.google.com -p 443
```

This fails with error -188 because Google's CA certificate wasn't loaded with the "-A" command line option.

F.3.2 WANT_READ (-323)

The WANT_READ error happens often when using non-blocking sockets, and isn't actually an error when using non-blocking sockets, but it is passed up to the caller as an error. When a call to receive data from the I/O callback would block as there isn't data currently available to receive, the I/O callback returns WANT_READ. The caller should wait and try receiving again later. This is usually seen from calls to wolfSSL_read(), wolfSSL_negotiate(), wolfSSL_accept(), and wolfSSL_connect(). The example client and server will indicate the WANT_READ incidents when debugging is enabled.

G Supported Cipher Suites, Key Exchange Groups, and Signature Algorithms

G.1 A. Cipher Suites

G.1.1 A-1. TLS 1.3

IANA Name	Wire Value	Default	configure Option
TLS_AES_128_GCM_SHA256	0x13,0x01	**Enabled**	--enable-tls13 + --enable-aesgcm
TLS_AES_256_GCM_SHA384	0x13,0x02	**Enabled**	--enable-tls13 + --enable-aesgcm + --enable-sha384
TLS_CHACHA20_POLY1305_SHA256	0x13,0x03	**Enabled**	--enable-tls13 + --enable-chacha + --enable-poly1305
TLS_AES_128_CCM_SHA256	0x13,0x04	Disabled	--enable-aesccm
TLS_AES_128_CCM_8_SHA256	0x13,0x05	Disabled	--enable-aesccm
TLS_SM4_GCM_SM3	0x00,0xC6	Disabled	--enable-sm4-gcm + --enable-sm3
TLS_SM4_CCM_SM3	0x00,0xC7	Disabled	--enable-sm4-ccm + --enable-sm3
TLS_SHA256_SHA256	0xC0,0xB4	Disabled	--enable-nullcipher
TLS_SHA384_SHA384	0xC0,0xB5	Disabled	--enable-nullcipher + --enable-sha384

G.1.2 A-2. TLS 1.2 ECDHE-ECDSA

IANA Name	Wire Value	Default	configure Option
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256	0xC0,0x2B	**Enabled**	—
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384	0xC0,0x2C	**Enabled**	—
TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256	0xCC,0xA9	**Enabled**	—
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA	0xC0,0x09	**Enabled**	—
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA	0xC0,0x0A	**Enabled**	—
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256	0xC0,0x23	**Enabled**	—
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384	0xC0,0x24	**Enabled**	--enable-sha384
TLS_ECDHE_ECDSA_WITH_AES_128_CCM	0xC0,0xAC	Disabled	--enable-aesccm
TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8	0xC0,0xAE	Disabled	--enable-aesccm
TLS_ECDHE_ECDSA_WITH_AES_256_CCM_8	0xC0,0xAF	Disabled	--enable-aesccm
TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA	0xC0,0x08	Disabled	--enable-des3
TLS_ECDHE_ECDSA_WITH_RC4_128_SHA	0xC0,0x07	Disabled	--enable-arc4
TLS_ECDHE_ECDSA_WITH_NULL_SHA	0xC0,0x06	Disabled	--enable-nullcipher
TLS_ECDHE_ECDSA_WITH_ARIA_128_GCM_SHA256	0xC0,0x5C	Disabled	--enable-aria
TLS_ECDHE_ECDSA_WITH_ARIA_256_GCM_SHA384	0xC0,0x5D	Disabled	--enable-aria
TLS_ECDHE_ECDSA_WITH_SM4_CBC_SM3	0xE0,0x11	Disabled	--enable-sm4-cbc + --enable-sm3 + --enable-sm2
TLS_ECDHE_ECDSA_WITH_SM4_GCM_SM3	0xE0,0x51	Disabled	--enable-sm4-gcm + --enable-sm3 + --enable-sm2

IANA Name	Wire Value	Default	configure Option
TLS_ECDHE_ECDSA_WITH_SM4_GCM_SM3	0xC0,0x52	Disabled	--enable-sm4-gcm + --enable-sm3 + --enable-sm2

G.1.3 A-3. TLS 1.2 ECDHE-RSA

IANA Name	Wire Value	Default	configure Option
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256	0xC0,0x2F	**Enabled**	—
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384	0xC0,0x30	**Enabled**	—
TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256	0xCC,0xA8	**Enabled**	—
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA	0xC0,0x13	**Enabled**	—
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA	0xC0,0x14	**Enabled**	—
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256	0xC0,0x27	**Enabled**	—
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384	0xC0,0x28	**Enabled**	--enable-sha384
TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA	0xC0,0x12	Disabled	--enable-des3
TLS_ECDHE_RSA_WITH_RC4_128_SHA	0xC0,0x11	Disabled	--enable-arc4

G.1.4 A-4. TLS 1.2 DHE-RSA

IANA Name	Wire Value	Default	configure Option
TLS_DHE_RSA_WITH_AES_128_GCM_SHA256	0x00,0x9E	Disabled	--enable-dh
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384	0x00,0x9F	Disabled	--enable-dh
TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256	0xCC,0xAA	Disabled	--enable-dh + --enable-chacha
TLS_DHE_RSA_WITH_AES_128_CBC_SHA	0x00,0x33	Disabled	--enable-dh
TLS_DHE_RSA_WITH_AES_256_CBC_SHA	0x00,0x39	Disabled	--enable-dh
TLS_DHE_RSA_WITH_AES_128_CBC_SHA256	0x00,0x67	Disabled	--enable-dh
TLS_DHE_RSA_WITH_AES_256_CBC_SHA256	0x00,0x6B	Disabled	--enable-dh
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA	0x00,0x16	Disabled	--enable-dh + --enable-des3
TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA	0x00,0x45	Disabled	--enable-dh + --enable-camellia
TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA	0x00,0x88	Disabled	--enable-dh + --enable-camellia
TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA256	0x00,0xBE	Disabled	--enable-dh + --enable-camellia
TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA256	0x00,0xC4	Disabled	--enable-dh + --enable-camellia

G.1.5 A-5. TLS 1.2 RSA (Static Key Exchange)

IANA Name	Wire Value	Default	configure Option
TLS_RSA_WITH_AES_128_GCM_SHA256	0x00,0x9C	**Enabled**	—
TLS_RSA_WITH_AES_256_GCM_SHA384	0x00,0x9D	**Enabled**	—
TLS_RSA_WITH_AES_128_CBC_SHA	0x00,0x2F	**Enabled**	—
TLS_RSA_WITH_AES_256_CBC_SHA	0x00,0x35	**Enabled**	—

COPYRIGHT ©2024 wolfSSL Inc.

IANA Name	Wire Value	Default	configure Option
TLS_RSA_WITH_AES_128_CBC_SHA256	0x00,0x3C	**Enabled**	—
TLS_RSA_WITH_AES_256_CBC_SHA256	0x00,0x3D	**Enabled**	—
TLS_RSA_WITH_AES_128_CCM_8	0xC0,0xA0	Disabled	--enable-aesccm
TLS_RSA_WITH_AES_256_CCM_8	0xC0,0xA1	Disabled	--enable-aesccm
SSL_RSA_WITH_3DES_EDE_CBC_SHA	0x00,0x0A	Disabled	--enable-des3
SSL_RSA_WITH_RC4_128_SHA	0x00,0x05	Disabled	--enable-arc4
SSL_RSA_WITH_RC4_128_MD5	0x00,0x04	Disabled	--enable-arc4
TLS_RSA_WITH_NULL_MD5	0x00,0x01	Disabled	--enable-nullcipher
TLS_RSA_WITH_NULL_SHA	0x00,0x02	Disabled	--enable-nullcipher
TLS_RSA_WITH_NULL_SHA256	0x00,0x3B	Disabled	--enable-nullcipher
TLS_RSA_WITH_CAMELLIA_128_CBC_SHA	0x00,0x41	Disabled	--enable-camellia
TLS_RSA_WITH_CAMELLIA_256_CBC_SHA	0x00,0x84	Disabled	--enable-camellia
TLS_RSA_WITH_CAMELLIA_128_CBC_SHA256	0x00,0xBA	Disabled	--enable-camellia
TLS_RSA_WITH_CAMELLIA_256_CBC_SHA256	0x00,0xC0	Disabled	--enable-camellia

G.1.6 A-6. TLS 1.2 Static ECDH

IANA Name	Wire Value	Default	configure Option
TLS_ECDH_RSA_WITH_AES_128_GCM_SHA256	0xC0,0x31	Disabled	--enable-staticecdh
TLS_ECDH_RSA_WITH_AES_256_GCM_SHA384	0xC0,0x32	Disabled	--enable-staticecdh
TLS_ECDH_RSA_WITH_AES_128_CBC_SHA	0xC0,0x0E	Disabled	--enable-staticecdh
TLS_ECDH_RSA_WITH_AES_256_CBC_SHA	0xC0,0x0F	Disabled	--enable-staticecdh
TLS_ECDH_RSA_WITH_AES_128_CBC_SHA256	0xC0,0x29	Disabled	--enable-staticecdh
TLS_ECDH_RSA_WITH_AES_256_CBC_SHA384	0xC0,0x2A	Disabled	--enable-staticecdh
TLS_ECDH_RSA_WITH_RC4_128_SHA	0xC0,0x0C	Disabled	--enable-staticecdh + --enable-arc4
TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA	0xC0,0x0D	Disabled	--enable-staticecdh + --enable-des3
TLS_ECDH_ECDSA_WITH_AES_128_GCM_SHA256	0xC0,0x2D	Disabled	--enable-staticecdh
TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA384	0xC0,0x2E	Disabled	--enable-staticecdh
TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA	0xC0,0x04	Disabled	--enable-staticecdh
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA	0xC0,0x05	Disabled	--enable-staticecdh
TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256	0xC0,0x25	Disabled	--enable-staticecdh
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA384	0xC0,0x26	Disabled	--enable-staticecdh
TLS_ECDH_ECDSA_WITH_RC4_128_SHA	0xC0,0x02	Disabled	--enable-staticecdh + --enable-arc4
TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA	0xC0,0x03	Disabled	--enable-staticecdh + --enable-des3

G.1.7 A-7. TLS 1.2 PSK

IANA Name	Wire Value	Default	configure Option
TLS_PSK_WITH_AES_128_GCM_SHA256	0x00,0xA8	Disabled	--enable-psk
TLS_PSK_WITH_AES_256_GCM_SHA384	0x00,0xA9	Disabled	--enable-psk
TLS_PSK_WITH_AES_128_CBC_SHA256	0x00,0xAE	Disabled	--enable-psk
TLS_PSK_WITH_AES_256_CBC_SHA384	0x00,0xAF	Disabled	--enable-psk
TLS_PSK_WITH_AES_128_CBC_SHA	0x00,0x8C	Disabled	--enable-psk
TLS_PSK_WITH_AES_256_CBC_SHA	0x00,0x8D	Disabled	--enable-psk

IANA Name	Wire Value	Default	configure Option
TLS_PSK_WITH_AES_128_CCM	0xC0,0xA4	Disabled	--enable-psk + --enable-aesccm
TLS_PSK_WITH_AES_256_CCM	0xC0,0xA5	Disabled	--enable-psk + --enable-aesccm
TLS_PSK_WITH_AES_128_CCM_8	0xC0,0xA8	Disabled	--enable-psk + --enable-aesccm
TLS_PSK_WITH_AES_256_CCM_8	0xC0,0xA9	Disabled	--enable-psk + --enable-aesccm
TLS_PSK_WITH_CHACHA20_POLY1305_SHA256	0xCC,0xAB	Disabled	--enable-psk + --enable-chacha
TLS_PSK_WITH_NULL_SHA256	0x00,0xB0	Disabled	--enable-psk + --enable-nullcipher
TLS_PSK_WITH_NULL_SHA384	0x00,0xB1	Disabled	--enable-psk + --enable-nullcipher
TLS_PSK_WITH_NULL_SHA	0x00,0x2C	Disabled	--enable-psk + --enable-nullcipher

G.1.8 A-8. TLS 1.2 DHE-PSK

IANA Name	Wire Value	Default	configure Option
TLS_DHE_PSK_WITH_AES_128_GCM_SHA256	0x00,0xAA	Disabled	--enable-psk + --enable-dh
TLS_DHE_PSK_WITH_AES_256_GCM_SHA384	0x00,0xAB	Disabled	--enable-psk + --enable-dh
TLS_DHE_PSK_WITH_AES_128_CBC_SHA256	0x00,0xB2	Disabled	--enable-psk + --enable-dh
TLS_DHE_PSK_WITH_AES_256_CBC_SHA384	0x00,0xB3	Disabled	--enable-psk + --enable-dh
TLS_DHE_PSK_WITH_AES_128_CCM	0xC0,0xA6	Disabled	--enable-psk + --enable-dh + --enable-aesccm
TLS_DHE_PSK_WITH_AES_256_CCM	0xC0,0xA7	Disabled	--enable-psk + --enable-dh + --enable-aesccm
TLS_DHE_PSK_WITH_CHACHA20_POLY1305_SHA256	0xCC,0xAD	Disabled	--enable-psk + --enable-dh + --enable-chacha
TLS_DHE_PSK_WITH_NULL_SHA256	0x00,0xB4	Disabled	--enable-psk + --enable-dh + --enable-nullcipher
TLS_DHE_PSK_WITH_NULL_SHA384	0x00,0xB5	Disabled	--enable-psk + --enable-dh + --enable-nullcipher

G.1.9 A-9. TLS 1.2 ECDHE-PSK

IANA Name	Wire Value	Default	configure Option
TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256	0xC0,0x37	Disabled	--enable-psk + --enable-supportedcurves
TLS_ECDHE_PSK_WITH_AES_128_GCM_SHA256	0xC0,0x38	Disabled	--enable-psk + --enable-supportedcurves
TLS_ECDHE_PSK_WITH_CHACHA20_POLY1305_SHA256	0xCC,0xAC	Disabled	--enable-psk + --enable-chacha
TLS_ECDHE_PSK_WITH_NULL_SHA256	0x00,0x3A	Disabled	--enable-psk + --enable-nullcipher

G.1.10 A-10. TLS 1.2 DH_anon

IANA Name	Wire Value	Default	configure Option
TLS_DH_anon_WITH_AES_128_CBC_SHA	0x00,0x34	Disabled	--enable-anon
TLS_DH_anon_WITH_AES_256_GCM_SHA384	0x00,0xA7	Disabled	--enable-anon

---

G.2 B. Key Exchange Groups

G.2.1 B-1. NIST Elliptic Curves

Group Name	ID	Default	configure Option
secp256r1 (P-256)	23	**Enabled**	--enable-ecc
secp384r1 (P-384)	24	**Enabled**	--enable-ecc
secp521r1 (P-521)	25	**Enabled**	--enable-ecc
secp192r1 (P-192)	19	Disabled	--enable-ecc (TLS 1.2 only)
secp224r1 (P-224)	21	Disabled	--enable-ecc (TLS 1.2 only)

G.2.2 B-2. Koblitz Curves

Group Name	ID	Default	configure Option
secp160k1	15	Disabled	--enable-ecccustcurves
secp192k1	18	Disabled	--enable-ecccustcurves
secp224k1	20	Disabled	--enable-ecccustcurves
secp256k1	22	Disabled	--enable-ecccustcurves

G.2.3 B-3. Brainpool Curves

Group Name	ID	TLS Ver.	Default	configure Option
brainpoolP256r1	26	1.2	Disabled	--enable-ecccustcurves + --enable-brainpool
brainpoolP384r1	27	1.2	Disabled	--enable-ecccustcurves + --enable-brainpool

COPYRIGHT ©2024 wolfSSL Inc.

Group Name	ID	TLS Ver.	Default	configure Option
brainpoolP512r1	28	1.2	Disabled	--enable-ecccustcurves + --enable-brainpool
brainpoolP256r1tls13	31	1.3	Disabled	--enable-ecccustcurves + --enable-brainpool
brainpoolP384r1tls13	32	1.3	Disabled	--enable-ecccustcurves + --enable-brainpool
brainpoolP512r1tls13	33	1.3	Disabled	--enable-ecccustcurves + --enable-brainpool

G.2.4 B-4. X25519 / X448

Group Name	ID	Default	configure Option
x25519	29	Disabled	--enable-curve25519
x448	30	Disabled	--enable-curve448

G.2.5 B-5. FFDHE

Group Name	ID	Default	configure Option
ffdhe2048	256	Conditionally Enabled	--enable-dh (Automatically enabled when TLS 1.3 + DH is enabled)
ffdhe3072	257	Disabled	-DHAVE_FFDHE_3072
ffdhe4096	258	Disabled	-DHAVE_FFDHE_4096
ffdhe6144	259	Disabled	-DHAVE_FFDHE_6144
ffdhe8192	260	Disabled	-DHAVE_FFDHE_8192

G.2.6 B-6. SM2

Group Name	ID	Default	configure Option
sm2p256v1	41	Disabled	--enable-sm2

G.2.7 B-7. ML-KEM Standalone

Group Name	ID	Default	configure Option
ML-KEM-512	512	Disabled	--enable-mlkem + --enable-tls-mlkem-standalone
ML-KEM-768	513	Disabled	--enable-mlkem + --enable-tls-mlkem-standalone
ML-KEM-1024	514	Disabled	--enable-mlkem + --enable-tls-mlkem-standalone

G.2.8 B-8. ML-KEM Hybrid (Standard)

Group Name	ID	Default	configure Option
SecP256r1MLKEM768	4587	Conditionally Enabled	--enable-mlkem + --enable-pqc-hybrids
X25519MLKEM768	4588	Disabled	--enable-mlkem + --enable-pqc-hybrids + --enable-curve25519
SecP384r1MLKEM1024	4589	Conditionally Enabled	--enable-mlkem + --enable-pqc-hybrids

G.2.9 B-9. ML-KEM Hybrid (Experimental)

Group Name	ID	Default	configure Option
SecP256r1MLKEM512	12107	Disabled	--enable-extra-pqc-hybrids + --enable-experimental
SecP384r1MLKEM768	12108	Disabled	--enable-extra-pqc-hybrids + --enable-experimental
SecP521r1MLKEM1024	12109	Disabled	--enable-extra-pqc-hybrids + --enable-experimental
X25519MLKEM512	12214	Disabled	Same as above + --enable-curve25519
X448MLKEM768	12215	Disabled	Same as above + --enable-curve448

G.2.10 B-10. Kyber Round 3 Legacy

Group Name	ID	Default	configure Option
KYBER_LEVEL1	570	Disabled	--enable-mlkem=original
KYBER_LEVEL3	572	Disabled	--enable-mlkem=original
KYBER_LEVEL5	573	Disabled	--enable-mlkem=original
P256_KYBER_LEVEL1	12090	Disabled	--enable-mlkem=original
P384_KYBER_LEVEL3	12092	Disabled	--enable-mlkem=original
P521_KYBER_LEVEL5	12093	Disabled	--enable-mlkem=original
X25519_KYBER_LEVEL1	12089	Disabled	Same as above + --enable-curve25519
X25519_KYBER_LEVEL3	25497	Disabled	Same as above + --enable-curve25519
X448_KYBER_LEVEL3	12176	Disabled	Same as above + --enable-curve448
P256_KYBER_LEVEL3	25498	Disabled	--enable-mlkem=original

G.3 C. Signature Algorithms

G.3.1 C-1. ECDSA (NIST Curves)

IANA Name	Wire Value	Default	configure Option
ecdsa_secp256r1_sha256	0x04,0x03	**Enabled**	--enable-ecc
ecdsa_secp384r1_sha384	0x05,0x03	**Enabled**	--enable-ecc + --enable-sha384
ecdsa_secp521r1_sha512	0x06,0x03	**Enabled**	--enable-ecc + --enable-sha512

IANA Name	Wire Value	Default	configure Option
ecdsa_sha1	0x02,0x03	Conditional	!NO_OLD_TLS or WOLFSSL_ALLOW_TLS_SHA1

G.3.2 C-2. ECDSA (Brainpool, TLS 1.3 Only)

IANA Name	Wire Value	Default	configure Option
ecdsa_brainpoolP256r1tls13_sha256	0x08,0x1A	Disabled	--enable-ecccustcurves + --enable-brainpool
ecdsa_brainpoolP384r1tls13_sha384	0x08,0x1B	Disabled	Same as above + --enable-sha384
ecdsa_brainpoolP512r1tls13_sha512	0x08,0x1C	Disabled	Same as above + --enable-sha512

G.3.3 C-3. EdDSA

IANA Name	Wire Value	Default	configure Option
ed25519	0x08,0x07	Disabled	--enable-ed25519
ed448	0x08,0x08	Disabled	--enable-ed448

G.3.4 C-4. RSA-PSS RSAE

IANA Name	Wire Value	Default	configure Option
rsa_pss_rsae_sha256	0x08,0x04	**Enabled**	--enable-rsapss (automatic with TLS 1.3)
rsa_pss_rsae_sha384	0x08,0x05	**Enabled**	--enable-rsapss + --enable-sha384
rsa_pss_rsae_sha512	0x08,0x06	**Enabled**	--enable-rsapss + --enable-sha512

G.3.5 C-5. RSA-PSS PSS (TLS 1.3 Only)

IANA Name	Wire Value	Default	configure Option
rsa_pss_pss_sha256	0x08,0x09	**Enabled**	--enable-tls13 + --enable-rsapss
rsa_pss_pss_sha384	0x08,0x0A	**Enabled**	Same as above + --enable-sha384
rsa_pss_pss_sha512	0x08,0x0B	**Enabled**	Same as above + --enable-sha512

G.3.6 C-6. RSA PKCS#1 v1.5

IANA Name	Wire Value	Default	configure Option
rsa_pkcs1_sha256	0x04,0x01	**Enabled**	—
rsa_pkcs1_sha384	0x05,0x01	**Enabled**	--enable-sha384
rsa_pkcs1_sha512	0x06,0x01	**Enabled**	--enable-sha512

COPYRIGHT ©2024 wolfSSL Inc.

IANA Name	Wire Value	Default	configure Option
rsa_pkcs1_sha224	0x03,0x01	Conditional	--enable-sha224 (Enabled by default on x86_64/aarch64)
rsa_pkcs1_sha1	0x02,0x01	Conditional	!NO_OLD_TLS or WOLFSSL_ALLOW_TLS_SHA1

G.3.7 C-7. SM2

IANA Name	Wire Value	Default	configure Option
sm2sig_sm3	0x07,0x08	Disabled	--enable-sm2 + --enable-sm3

G.3.8 C-8. ML-DSA / Dilithium

IANA Name	Wire Value	Default	configure Option
mldsa44 (ML-DSA-44)	0x09,0x04	Disabled	--enable-mldsa
mldsa65 (ML-DSA-65)	0x09,0x05	Disabled	--enable-mldsa
mldsa87 (ML-DSA-87)	0x09,0x06	Disabled	--enable-mldsa

G.4 D. configure Option → #define Reference

configure Option	#define Macro	Notes
--enable-tls13	WOLFSSL_TLS13	Enabled by default
--enable-ecc	HAVE_ECC	Enabled by default
--enable-aesgcm	HAVE_AESGCM	Enabled by default
--enable-chacha	HAVE_CHACHA	Enabled by default
--enable-poly1305	HAVE_POLY1305	Enabled by default
--enable-sha384	WOLFSSL_SHA384	Enabled by default
--enable-sha512	WOLFSSL_SHA512	Enabled by default
--enable-supportedcurves	HAVE_SUPPORTED_CURVES + HAVE_TLS_EXTENSIONS	Enabled by default
--enable-aesccm	HAVE_AESCCM	
--enable-dh	HAVE_DH	
--enable-psk	HAVE_PSK	
--enable-rsapss	WC_RSA_PSS	Automatic when TLS 1.3 is enabled
--enable-curve25519	HAVE_CURVE25519	
--enable-curve448	HAVE_CURVE448	

configure Option	#define Macro	Notes
--enable-ed25519	HAVE_ED25519	
--enable-ed448	HAVE_ED448	
--enable-ecccustcurves	WOLFSSL_CUSTOM_CURVES	
--enable-brainpool	HAVE_ECC_BRAINPOOL	Requires ecccustcurves
--enable-des3	Removes NO_DES3	
--enable-arc4	Removes NO_RC4	
--enable-camellia	HAVE_CAMELLIA	
--enable-aria	HAVE_ARIA	
--enable-nullcipher	HAVE_NULL_CIPHER	Deprecated
--enable-anon	HAVE_ANON	Deprecated
--enable-sm2	WOLFSSL_SM2	
--enable-sm3	WOLFSSL_SM3	
--enable-sm4-gcm	WOLFSSL_SM4_GCM	
--enable-mlkem	WOLFSSL_HAVE_MLKEM + HAVE_PQC	
--enable-pqc-hybrids	WOLFSSL_PQC_HYBRIDS	Enabled by default when mlkem is enabled
--enable-mldsa	HAVE_DILITHIUM	
--enable-staticecdh	WOLFSSL_STATIC_ECDH	No PFS, deprecated

H wolfSSL Hardware Optimization Support Reference

H.1 A. Architecture-Level ISA/SIMD Optimizations

Architecture	Algorithms	user_settings.h Macro	configure Option	Notes
Intel x86-64	AES (all modes)	`WOLFSSL_AESNI`	`--enable-aesni`	AES-NI instructions
Intel x86-64	SHA-256, SHA-512	`USE_INTEL_SPEEDUP` + `HAVE_INTEL_AVX1`	Auto-detected	AVX1 assembly
Intel x86-64	SHA-512, SHA-3	`USE_INTEL_SPEEDUP` + `HAVE_INTEL_AVX2`	Auto-detected	AVX2 assembly
Intel x86-64	ChaCha20, Poly1305	`USE_INTEL_SPEEDUP`	Auto-detected	
Intel x86-64	X25519 (Curve25519), Ed25519	`USE_INTEL_SPEEDUP`	Auto-detected	fe_x25519_asm.S
Intel x86-64	RSA/ECC/DH (SP)	`WOLFSSL_SP_X86_64_ASM`	Auto-detected	sp_x86_64_asm.S
ARM AArch64	AES, SHA-256/512, SHA-3, ChaCha20, Poly1305	`WOLFSSL_ARMASM`	`--enable-armasm`	ARMv8 CE (Crypto Extension)
ARM AArch64	X25519 (Curve25519), Ed25519	`WOLFSSL_ARMASM`	`--enable-armasm`	armv8-curve25519.S
ARM AArch64	RSA/ECC/DH (SP)	`WOLFSSL_SP_ARM64_ASM`	Auto-detected	sp_arm64.c
ARM 32-bit (ARMv8-32)	AES, SHA-256/512, SHA-3, ChaCha20, Poly1305	`WOLFSSL_ARMASM`	`--enable-armasm`	
ARM 32-bit	X25519 (Curve25519), Ed25519	`WOLFSSL_ARMASM`	`--enable-armasm`	armv8-32-curve25519.S
ARM 32-bit	RSA/ECC/DH (SP)	`WOLFSSL_SP_ARM32_ASM`	Auto-detected	sp_arm32.c
ARM Thumb2	X25519 (Curve25519), Ed25519	`WOLFSSL_ARMASM` + `WOLFSSL_ARMASM_THUMB2`	`--enable-armasm`	thumb2-curve25519.S
ARM Thumb	RSA/ECC/DH (SP)	`WOLFSSL_SP_ARM_THUMB_ASM`	Auto-detected	sp_armthumb.c
ARM Cortex-M	RSA/ECC/DH (SP)	`WOLFSSL_SP_ARM_CORTEX_ASM`	Auto-detected	sp_cortexm.c
RISC-V 64-bit	AES, SHA-256/512, SHA-3, ChaCha20, Poly1305	`WOLFSSL_RISCV_ASM`	`--enable-riscv`	riscv/riscv-64-*.c
PowerPC 32-bit	SHA-256	`WOLFSSL_PPC32_ASM`	Auto-detected	ppc32/ppc32-sha256-asm.S

H.2 B. SP Math Library (Public Key Cryptography)

Architecture	Algorithms	user_settings.h Macro	configure Option
x86-64 assembly	RSA 2048/3072/4096, ECC P-256/P-384/P-521, DH	WOLFSSL_SP_X86_64_ASM	Auto-detected
ARM AArch64 assembly	RSA, ECC, DH	WOLFSSL_SP_ARM64_ASM	Auto-detected
ARM 32-bit assembly	RSA, ECC, DH	WOLFSSL_SP_ARM32_ASM	Auto-detected
ARM Thumb assembly	RSA, ECC, DH	WOLFSSL_SP_ARM_THUMB_ASM	Auto-detected
Cortex-M assembly	RSA, ECC, DH	WOLFSSL_SP_ARM_CORTEX_M_ASM	Auto-detected
Generic C (32-bit / 64-bit)	RSA, ECC, DH	WOLFSSL_SP_MATH	--enable-sp

H.2.1 sp_int.c Inline Assembly Speedups

sp_int.c contains small inline assembly helpers for multi-precision multiply/add operations used when the dedicated per-algorithm SP files (e.g. sp_x86_64.c) are absent or for non-RSA/ECC key sizes. These are controlled by the following macros (auto-detected; can also be set in user_settings.h):

Architecture	user_settings.h Macro
Intel x86-64	WOLFSSL_SP_X86_64
Intel x86 (32-bit)	WOLFSSL_SP_X86
ARM AArch64	WOLFSSL_SP_ARM64
ARM 32-bit	WOLFSSL_SP_ARM32
ARM Thumb	WOLFSSL_SP_ARM_THUMB
RISC-V 64-bit	WOLFSSL_SP_RISCV64
RISC-V 32-bit	WOLFSSL_SP_RISCV32
MIPS 64-bit	WOLFSSL_SP_MIPS64
MIPS 32-bit	WOLFSSL_SP_MIPS
s390x	WOLFSSL_SP_S390X

H.2.2 SP Key Size Configuration

```
#define WOLFSSL_SP_MATH
#define WOLFSSL_SP_X86_64_ASM         /* x86-64 */
#define WOLFSSL_SP_ARM64_ASM          /* AArch64 */
#define WOLFSSL_SP_ARM_CORTEX_M_ASM   /* Cortex-M */

#define WOLFSSL_SP_2048  /* RSA-2048 */
#define WOLFSSL_SP_3072  /* RSA-3072 */
#define WOLFSSL_SP_4096  /* RSA-4096 */
#define WOLFSSL_SP_256   /* ECC P-256 */
#define WOLFSSL_SP_384   /* ECC P-384 */
#define WOLFSSL_SP_521   /* ECC P-521 */
```

H.3 C. Microcontroller/SoC Chip-Specific Hardware Accelerators

Vendor / Chip	Algorithms	user_settings.h Macro	configure Option	Source
STMicroelectronics				
STM32F2/F4/F7/H7 (CRYP)	AES-ECB/CBC/CTR/GCM	STM32_CRYPTO	HAL auto-detected	port/st/stm32.c
STM32L4/L5/WB55	AES, SHA (partial)	STM32_CRYPTO	HAL auto-detected	port/st/stm32.c
STM32U5 (DHUK)	AES with key wrapping	WOLFSSL_STM32U5_DHUK	HAL auto-detected	port/st/stm32.c
STM32MP1/MP13/MP25	AES, SHA, RSA, ECC	WOLFSSL_STM32_CRYPT0	HAL auto-detected	port/st/stm32.c
NXP				
i.MX RT (DCP)	AES-ECB/CBC/CTR/GCM, SHA	WOLFSSL_IMXRT_DCP	Auto-detected	port/nxp/dcp_port.c
i.MX 6/8 (CAAM)	AES, SHA, RSA, ECC, DH, HMAC	WOLFSSL_IMX6_CAAM / WOLFSSL_SECO_CAAM	--enable-caam	port/caam/*
SE050 Secure Element	AES, SHA, RSA, ECC, X25519, ECDH	WOLFSSL_SE050 + WOLFSSL_SE050_CRYPT	Auto-detected	port/nxp/se050_port.c
Espressif				
ESP32 / ESP32-S2 / ESP32-S3	AES, SHA-1/256/384/512, RSA, ECC	WOLFSSL_ESP32_CRYPT	--enable-esp32	port/Espressif/esp32_*.c
ESP32-C3 / C6 / H2	AES, SHA-256/384/512 (partial)	WOLFSSL_ESP32_CRYPT	--enable-esp32	port/Espressif/esp32_*.c
Renesas				
RX64M/RX71M/RX72M	SHA-1, SHA-256	WOLFSSL_RENESAS_RX64_HASH	Auto-detected	port/Renesas/renesas_rx64_*.c
RX65N (SCE)	AES, SHA, RSA, ECC, RNG	WOLFSSL_RENESAS_SCEPROTECT	Auto-detected	port/Renesas/renesas_*.c
RZ/A1/A2 (TSIP)	AES, SHA, RSA, ECC, RNG	WOLFSSL_RENESAS_TSIP	Auto-detected	port/Renesas/renesas_tsip_*.c
RA4M4/RA6M4 (RSIP)	AES, SHA, RSA, ECC, RNG	WOLFSSL_RENESAS_RSIP	Auto-detected	port/Renesas/renesas_rsip*.c
RA FSP Secure Manager	AES, SHA, RSA, ECC, HMAC	WOLFSSL_RENESAS_FSPSM	Auto-detected	port/Renesas/renesas_fspsm_*.c
Cypress				
PSoC 6	AES, SHA, ECC, RNG	WOLFSSL_PSOC6_CRYPTO	Auto-detected	port/cypress/psoc6_crypto.c
Silicon Labs				
EFR32 / EFM32 (SE)	AES, SHA, ECC, RNG	WOLFSSL_SILABS_SE_ACCEL	Auto-detected	port/silabs/silabs_*.c
Maxim				
MAX3266X	AES-ECB/CBC, RSA	WOLFSSL_MAX3266X	Auto-detected	port/maxim/max3266x.c
MAXQ10XX	AES, SHA, RSA, ECC, RNG	WOLFSSL_MAXQ10XX_CRYPTO	--with-maxq10xx=PART	port/maxim/maxq10xx.c
Microchip				
PIC32MZ	AES, SHA, RNG	WOLFSSL_PIC32MZ_CRYPT	Auto-detected	port/pic32/
ATECC508A / 608A	AES, SHA, ECC (ECDH/ECDSA)	WOLFSSL_ATECC	Auto-detected	port/atmel/atmel.c

Vendor / Chip	Algorithms	user_settings.h Macro	configure Option	Source
Xilinx				
Zynq / Zynq-7000	AES, SHA, RSA, ECC	`WOLFSSL_XILINX_CRYPT`	`--enable-xilinx`	port/xilinx/xil-*.c
Versal	AES-GCM, SHA-3, RNG	`WOLFSSL_XILINX_CRYPT_VERSAL`	`--enable-xilinx`	port/xilinx/xil-versal-*.c
ARM CryptoCell				
CC-300/312/700/712/714	AES, SHA, RSA, ECC, ChaCha, Poly1305	`WOLFSSL_CRYPTOCELL`	`--enable-cryptocell`	port/arm/cryptocell*.c
IoT-Safe (SIM)	ECC, SHA, RNG	`WOLFSSL_IOTSAFE`	`--enable-iotsafe`	port/iotsafe/

H.4 D. External Accelerators and Coprocessors

Product	Algorithms	user_settings.h Macro	configure Option
Intel QuickAssist (QAT)	AES, RSA, DH, ECC, SHA	`HAVE_INTEL_QA`	`--with-intelqa=PATH`
Intel QAT (synchronous)	Same as above	`ENABLED_INTEL_QA_SYNC`	`--with-intelqa-sync=PATH`
Cavium NITROX	AES, RSA, DH, ECC, SHA	`HAVE_CAVIUM`	`--with-cavium=PATH`
Cavium NITROX V	Same as above	`HAVE_CAVIUM_V`	`--with-cavium=PATH`
Cavium Octeon (MIPS)	AES, RSA, ECC, SHA	`HAVE_CAVIUM_OCTEON`	`--with-octeon-sync=PATH`
NXP CAAM	AES, SHA, RSA, ECC, HMAC, DH	`WOLFSSL_CAAM`	`--enable-caam`

H.5 E. OS Kernel Hardware Offload Interfaces

Interface	Algorithms	user_settings.h Macro	configure Option
Linux `/dev/crypto`	AES, SHA-1/256/512, RSA, ECC, DH, HMAC	`WOLFSSL_DEVCRYPTO`	`--enable-devcrypto`
Linux Kernel Crypto API	AES, SHA, RSA, ECC, X25519, HMAC	`WOLFSSL_KCAPI`	`--enable-kcapi`
Linux AF_ALG	AES, SHA (kernel backend dependent)	`WOLFSSL_AFALG`	`--enable-afalg`
PSA (Platform Security Architecture)	Vendor implementation dependent	`WOLFSSL_HAVE_PSA`	`--enable-psa`

COPYRIGHT ©2024 wolfSSL Inc.

I Experimenting with Post-Quantum Cryptography

A while back, the wolfSSL team integrated experimental post-quantum cryptographic algorithms into the wolfSSL library. This was done by integrating with the Open Quantum Safe team's liboqs. Currently, wolfCrypt implements LMS, XMSS, ML-DSA and ML-KEM. So, for the purpose of code size reduction and ease of maintenance, the wolfSSL team removed the integration with liboqs.

This appendix is intended for anyone that wants to start learning about post-quantum cryptography in the context of (D)TLS 1.3. It explains why post-quantum algorithms are important, what we have done in response to the quantum threat and how you can start experimenting with these new algorithms.

Note: Some of the post-quantum algorithms are not fully standardized yet. Some OIDs and codepoints are temporary and expected to change in the future. You should have no expectation of backwards compatibility until they are fully standardized.

I.1 A Gentle Introduction to Post-Quantum Cryptography

I.1.1 Why Post-Quantum Cryptography?

For some time now, many resources have been devoted to the development of quantum computers. So much so that commercialization of cloud quantum computing resources has already begun. While the current state of the art is still not in the realm of being cryptographically relevant, some threat models such as "harvest now, decrypt later" mean that preparations need to happen sooner than the appearance of cryptographically relevant quantum computers.

NIST is leading the way for standardization of a new class of algorithms designed to replace the public key cryptography algorithms that will become vulnerable to quantum computers. At the time of the writing of this passage, NIST has already standardized ML-DSA, ML-KEM, and SLH-DSA.

ML-KEM (Module Lattice Key Encapsulation Mechanism) is a NIST-standardized, lattice-based post-quantum algorithm derived from Kyber. It enables two parties to establish a shared key over an insecure channel using a key encapsulation mechanism, protecting against both classical and quantum adversaries.

ML-DSA (Module Lattice Digital Signature Algorithm) is a NIST-standardized, lattice-based post-quantum digital signature scheme derived from Dilithium. It enables a sender to produce a verifiable signature that proves the origin and integrity of a message.

Both ML-KEM and ML-DSA are public-key algorithms designed to resist cryptographically relevant quantum computers. They are part of NIST's Post-Quantum Cryptography standards (FIPS 203 and FIPS 204) and can be deployed today, often in hybrid form, to prepare for the post-quantum era.

Currently, standards organizations have various draft documents describing OIDs and codepoints. NIST is working on bringing these algorithms under the ubmbrella of the CMVP regulatory framework allowing for FIPS-140-3 validations of implementations of these algorithms.

I.1.2 How do we Protect Ourselves?

From a high level perspective, for every TLS 1.3 connection, authentication, integrity and confidentiality are the main security goals that protect each connection. Authentication is maintained via signature schemes such as ECDSA. Confidentiality and integrity are maintained by key establishment algorithms such as ECDHE and then using the established key with symmetric encryption algorithms such as AES to encrypt a communication stream. We can thus decompose the security of the TLS 1.3 protocol into 3 types of cryptographic algorithms:

- authentication algorithms
- key establishment algorithms

COPYRIGHT ©2024 wolfSSL Inc.

- symmetric cipher algorithms

The threat of quantum computers to conventional cryptography takes two forms. Grover's algorithm reduces the security of modern symmetric cipher algorithms by approximately half while Shor's algorithm completely breaks the security of modern authentication and key establishment algorithms. As a result, we can continue to protect our communications using the AES-256 symmetric cipher which is considered sufficiently secure even in the presence of a cryptographically relevant quantum computer. We can then replace our conventional authentication and key establishment algorithms with post-quantum algorithms. Note that during TLS 1.3 handshakes, the ciphersuite specifies the symmetric cipher to be used for the duration of the connection. Both CNSA (Commercial National Security Algorithm Suite) 1.0 and 2.0 prescribe using the AES_256_GCM_SHA384 ciphersuite. For key establishment and authentication, there are post-quantum KEMs (Key Encapsulation Mechanisms) and signature schemes.

These use different kinds of math from the conventional algorithms. They are designed specifically for resistance to quantum-computers. The authentication algorithm and KEM that NIST has standardized for use with network protocols are lattice-based algorithms.

- ML-DSA (Dilithium) Signature Scheme
- ML-KEM (KYBER) KEM

Note: SABER KEM and NTRU KEM were deprecated and removed as they did not move on to standardization.

Note: KYBER KEM 90s variants were deprecated and removed as NIST is not considering them for standardization.

Note: Dilithium Signature Scheme's AES variants were deprecated and removed as NIST is not considering them for standardization.

Note: When the liboqs integration was removed, we also removed the FALCON and SPHINCS+ signature schemes. We will have our own implementations in the future.

An explanation of lattice-based cryptography would fall outside the scope of this document but more information about these algorithms can be found in their NIST submissions at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

Unfortunately, it might come as a shock, but we do not actually know that these algorithms will resist attacks from quantum computers. In fact, we do not even know that these algorithms are safe against a conventional computer. Its getting less and less likely, but someone could break lattice-based cryptography. However, as security experts will tell you, this is how cryptography has always worked. Algorithms are good when we start using them, but weaknesses and vulnerabilities are discovered and technology gets better. The post-quantum algorithms are somewhat problematic in that they are relatively new and could use a bit more attention from the community.

One solution is to not put our full faith into these new algorithms. For now, we can hedge our bets by hybridizing post-quantum algorithms with the conventional algorithms that we actually trust. ECC with NIST standardized curves seem like good candidates as we have to keep using them since FIPS 140-3 compliance is a priority. For this reason, we have not only implemented post-quantum KEMs but also hybridized them with ECDH over NIST approved curves. Please see our list of hybrid groups below.

I.2 Getting Started with Post-Quantum algorithms in wolfSSL

The following instructions will get you started from a clean Linux development environment and lead you step by step to performing a quantum-safe TLS 1.3 connection.

I.2.1 Build Instructions

Please see the wolfSSL repo's INSTALL file (https://github.com/wolfSSL/wolfssl/blob/master/INSTALL). Item 15 has instructions on how to configure and build wolfSSL with ML-KEM and ML-DSA enabled.

You will need the patched OQS OpenSSL Provider fork in order to generate X.509 certificates with post-quantum cryptographic keys and signatures. Instructions can be found at https://github.com/wolfSSL/osp/tree/maste For your convenience, pre-generated certificates can be found there as well.

I.2.2 Making a Quantum Safe TLS Connection

You can run the server and client like this in separate terminals:

```
examples/server/server -v 4 -l TLS_AES_256_GCM_SHA384 \
      -A ../osp/oqs/mldsa87_root_cert.pem \
      -c ../osp/oqs/mldsa44_entity_cert.pem \
      -k ../osp/oqs/mldsa44_entity_key.pem \
      --pqc SecP521r1MLKEM1024
```

```
examples/client/client -v 4 -l TLS_AES_256_GCM_SHA384 \
      -A ../osp/oqs/mldsa44_root_cert.pem \
      -c ../osp/oqs/mldsa87_entity_cert.pem \
      -k ../osp/oqs/mldsa87_entity_key.pem \
      --pqc SecP521r1MLKEM1024
```

You have just achieved a fully quantum-safe TLS 1.3 connection using AES-256 for symmetric encryption, the ML-DSA signature scheme for authentication and ECDHE hybridized with ML-KEM for key establishment.

Further information about other post-quantum examples can be found at https://github.com/wolfSSL/wolfssl-examples/blob/master/pq/README.md.

I.3 Post Quantum Algorithm Variant Names

Post-Quantum algorithm variant names:

NIST Security Level	PQC Variant Name
2	ML_DSA_44
3	ML_DSA_65
5	ML_DSA_87
1	ML_KEM_512
3	ML_KEM_768
5	ML_KEM_1024

Post-Quantum hybrid KEM names:

wolfSSL Variant Name	NIST ECC Curve and PQC Submission Variant Name
SecP256r1MLKEM512	ECDSA P-256 and KYBER512
SecP384r1MLKEM768	ECDSA P-384 and KYBER768
SecP521r1MLKEM1024	ECDSA P-521 and KYBER1024

COPYRIGHT ©2024 wolfSSL Inc.

I.4 Cryptographic Artifact Sizes

All sizes are in bytes.

Post-Quantum Signature Scheme Artifact Sizes:

PQC Variant Name	Public Key Size	Private Key Size	Maximum Signature Size
ML_DSA_44	1312	2560	2420
ML_DSA_65	1952	4032	3309
ML_DSA_87	2592	4896	4627

Post-Quantum KEM Artifact Sizes:

PQC Variant Name	Public Key Size	Private Key Size	Ciphertext Size	Shared Secret Size
ML_KEM_512	800	1632	768	32
ML_KEM_768	1184	2400	1088	32
ML_KEM_1024	1568	3168	1568	32

I.5 Statistics

The following statistics and benchmarks were taken on an 11th GenIntel Core i7-1185G7@3-GHz with 8 cores running Ubuntu 22.04.5 LTS

wolfSSL:

```
./configure --enable-kyber \
            --enable-dilithium \
            --disable-psk \
            --disable-shared \
            --enable-intelasm \
            --enable-aesni \
            --enable-sp-math-all \
            --enable-sp-asm \
            CFLAGS="-Os"
```

Note: We are primarily benchmarking the post-quantum algorithms, but leave some conventional algorithms for comparison purposes.

I.5.1 Runtime Binary Sizes

The `tls_bench` example application binary file is 2498432 bytes after being built then stripped (Approximately 2.4M). Without `--enable-kyber --enable-dilithium` it is 2290912 bytes after being built then stripped (Approximately 2.2M). This is a difference of 207520 bytes (Approximately 200K).

I.5.2 TLS 1.3 Data Transmission Sizes

The following results were taken by running the example server and client and recording all information being transmitted via wireshark. This includes the TLS 1.3 handshake with mutual authentication, “hello wolfssl!” and “I hear you fa shizzle!” messages. The `tcp.len` of all packets were summed:

Ciphersuite	Authentication	Key Establishment	Total Bytes
TLS_AES_256_GCM_SHA384	RSA 2048 bit	ECC SECP256R1	5455
TLS_AES_256_GCM_SHA384	RSA 2048 bit	ML_KEM_512	6633
TLS_AES_256_GCM_SHA384	RSA 2048 bit	ML_KEM_768	7337
TLS_AES_256_GCM_SHA384	RSA 2048 bit	ML_KEM_1024	8201
TLS_AES_256_GCM_SHA384	RSA 2048 bit	SecP256r1MLKEM512	6763
TLS_AES_256_GCM_SHA384	RSA 2048 bit	SecP384r1MLKEM768	7531
TLS_AES_256_GCM_SHA384	RSA 2048 bit	SecP521r1MLKEM1024	8467
TLS_AES_256_GCM_SHA384	ML_DSA_44	ECC SECP256R1	7918
TLS_AES_256_GCM_SHA384	ML_DSA_65	ECC SECP256R1	10233
TLS_AES_256_GCM_SHA384	ML_DSA_87	ECC SECP256R1	13477

I.5.3 Heap and Stack Usage

NOTE: This is out of date. These were obtained when wolfSSL was using the liboqs implementations of these algorithms. This is left here for historical purposes.

These statistics were obtained by adding the following configuration flags: `--enable-trackmemory --enable-stacksize`.

Memory use for server sign and client verify without server authentication of the client, TLS13-AES256-GCM-SHA384 ciphersuite and ECC SECP256R1 for key exchange.

```
Server FALCON_LEVEL1

stack used     =  48960
total Allocs   =    250
heap total     = 113548
heap peak      =  40990

Client FALCON_LEVEL1

stack used     =  29935
total Allocs   =    768
heap total     = 179427
heap peak      =  41765

Server FALCON_LEVEL5

stack used     =  89088
total Allocs   =    250
heap total     = 125232
heap peak      =  45630

Client FALCON_LEVEL5

stack used     =  29935
total Allocs   =    768
heap total     = 191365
heap peak      =  47469

Server DILITHIUM_LEVEL2

stack used = 56328
```

```
total   Allocs   =       243
total   Deallocs =       243
total   Bytes    =    128153
peak    Bytes    =     50250

Client DILITHIUM_LEVEL2

stack used = 30856
total   Allocs   =       805
total   Deallocs =       805
total   Bytes    =    206412
peak    Bytes    =     56299

Server DILITHIUM_LEVEL3

stack used = 86216
total   Allocs   =       243
total   Deallocs =       243
total   Bytes    =    140128
peak    Bytes    =     55161

Client DILITHIUM_LEVEL3

stack used = 33928
total   Allocs   =       805
total   Deallocs =       805
total   Bytes    =    220633
peak    Bytes    =     61245

Server DILITHIUM_LEVEL5

stack used = 119944
total   Allocs   =       243
total   Deallocs =       243
total   Bytes    =    152046
peak    Bytes    =     59829

Client DILITHIUM_LEVEL5

stack used = 40328
total   Allocs   =       805
total   Deallocs =       805
total   Bytes    =    238167
peak    Bytes    =     67049

Server RSA 2048

stack used     =  52896
total Allocs   =    253
heap total     = 121784
heap peak      =  39573

Client RSA 2048
```

```
stack used     =  54640
total Allocs   =    897
heap total     = 202472
heap peak      =  41760
```

Memory use for KEM groups. TLS13-AES256-GCM-SHA384 ciphersuite and RSA-2048 for client authentication of the server and without server authentication of the client.

```
Server KYBER_LEVEL1

stack used     = 52896
total Allocs   =   206
heap total     = 66864
heap peak      = 28474

Client KYBER_LEVEL1

stack used     =  54640
total Allocs   =    879
heap total     = 147235
heap peak      =  44538

Server KYBER_LEVEL3

stack used     =  52896
total Allocs   =    206
heap total     =  67888
heap peak      =  28794

Client KYBER_LEVEL3

stack used     =  54640
total Allocs   =    879
heap total     = 149411
heap peak      =  46010

Server KYBER_LEVEL5

stack used     =  52896
total Allocs   =    206
heap total     =  69232
heap peak      =  29274

Client KYBER_LEVEL5

stack used     =  54640
total Allocs   =    879
heap total     = 151907
heap peak      =  47642

Server KYBER_90S_LEVEL1

stack used     =  52896
total Allocs   =    206
heap total     =  66864
```

```
heap peak      =  28474

Client KYBER_90S_LEVEL1

stack used     =  54640
total Allocs   =    879
heap total     = 147235
heap peak      =  44538

Server KYBER_90S_LEVEL3

stack used     =  52896
total Allocs   =    206
heap total     =  67888
heap peak      =  28794

Client KYBER_90S_LEVEL3

stack used     =  54640
total Allocs   =    879
heap total     = 149411
heap peak      =  46010

Server KYBER_90S_LEVEL5

stack used     =  52896
total Allocs   =    206
heap total     =  69232
heap peak      =  29274

Client KYBER_90S_LEVEL5

stack used     =  54640
total Allocs   =    879
heap total     = 151907
heap peak      =  47642

Server P256_KYBER_LEVEL1

stack used     =  52896
total Allocs   =    223
heap total     = 118940
heap peak      =  37652

Client P256_KYBER_LEVEL1

stack used     =  54640
total Allocs   =    896
heap total     = 199376
heap peak      =  48932

Server P384_KYBER_LEVEL3

stack used     =  52896
```

```
total Allocs   =    223
heap total     = 120108
heap peak      =  38468

Client P384_KYBER_LEVEL3

stack used     =  54640
total Allocs   =    896
heap total     = 201728
heap peak      =  50468

Client Server P521_KYBER_LEVEL5

stack used     =  52896
total Allocs   =    223
heap total     = 121614
heap peak      =  39458

Client P521_KYBER_LEVEL5

stack used     =  54640
total Allocs   =    896
heap total     = 204422
heap peak      =  52172

Client Server P256_KYBER_90S_LEVEL1

stack used     =  52896
total Allocs   =    223
heap total     = 118940
heap peak      =  37652

Client P256_KYBER_90S_LEVEL1

stack used     =  54640
total Allocs   =    896
heap total     = 199376
heap peak      =  48932

Server P384_KYBER_90S_LEVEL3

stack used     =  52896
total Allocs   =    223
heap total     = 120108
heap peak      =  38468

Client P384_KYBER_90S_LEVEL3

stack used     =  54640
total Allocs   =    896
heap total     = 201728
heap peak      =  50468

Server P521_KYBER_90S_LEVEL5
```

```
stack used     =  52896
total Allocs   =    223
heap total     = 121614
heap peak      =  39458

Client P521_KYBER_90S_LEVEL5

stack used     =  54640
total Allocs   =    896
heap total     = 204422
heap peak      =  52172

Server ECDSA SECP256R1

stack used     =  52896
total Allocs   =    253
heap total     = 121784
heap peak      =  39573

Client ECDSA SECP256R1

stack used     =  54640
total Allocs   =    897
heap total     = 202472
heap peak      =  41760
```

I.5.4 Benchmarks

The following benchmarks were obtained with the following configuration flags:

```
./configure --enable-kyber \
            --enable-dilithium \
            --disable-shared \
            --enable-intelasm \
            --enable-aesni \
            --enable-sp \
            --enable-sp-math \
            --enable-sp-asm \
            CFLAGS="-Os -DECC_USER_CURVES -DHAVE_ECC256 -DHAVE_ECC384"
```

I.5.4.1 Benchmarks from wolfCrypt

Note: Only a single core is used.

```
CPU: Intel x86_64 - avx1 avx2 rdrand rdseed bmi2 aesni adx movbe bmi1 sha
Math:   Multi-Precision: Disabled
    Single Precision: ecc 256 384 521 rsa/dh 2048 3072 4096 asm sp_x86_64.c

ECC    SECP256R1 key gen 95600 ops took 1.000 sec, avg 0.010 ms, 95587.830 ops
   /sec
ECDHE  SECP256R1 agree   24800 ops took 1.003 sec, avg 0.040 ms, 24737.512 ops
   /sec
ECDSA  SECP256R1 sign    61400 ops took 1.001 sec, avg 0.016 ms, 61337.775 ops
   /sec
```

```
ECDSA  SECP256R1 verify  23000 ops took 1.001 sec, avg 0.044 ms, 22976.012 ops
    /sec
ML-KEM 512       key gen 284600 ops took 1.000 sec, avg 0.004 ms, 284565.467
    ops/sec
ML-KEM 512       encap   270800 ops took 1.000 sec, avg 0.004 ms, 270749.585
    ops/sec
ML-KEM 512       decap   172900 ops took 1.000 sec, avg 0.006 ms, 172896.249
    ops/sec
ML-KEM 768       key gen 159800 ops took 1.000 sec, avg 0.006 ms, 159776.306
    ops/sec
ML-KEM 768       encap   152800 ops took 1.000 sec, avg 0.007 ms, 152765.071
    ops/sec
ML-KEM 768       decap   100100 ops took 1.000 sec, avg 0.010 ms, 100091.147
    ops/sec
ML-KEM 1024      key gen 108300 ops took 1.000 sec, avg 0.009 ms, 108277.024
    ops/sec
ML-KEM 1024      encap   104400 ops took 1.000 sec, avg 0.010 ms, 104388.900
    ops/sec
ML-KEM 1024      decap   74100 ops took 1.001 sec, avg 0.014 ms, 74057.147 ops
    /sec
ML-DSA 44        key gen 20700 ops took 1.004 sec, avg 0.049 ms, 20617.041 ops
    /sec
ML-DSA 44        sign    5100 ops took 1.019 sec, avg 0.200 ms, 5003.233 ops/
    sec
ML-DSA 44        verify  18500 ops took 1.005 sec, avg 0.054 ms, 18403.134 ops
    /sec
ML-DSA 65        key gen 10200 ops took 1.007 sec, avg 0.099 ms, 10133.468 ops
    /sec
ML-DSA 65        sign    2900 ops took 1.004 sec, avg 0.346 ms, 2887.112 ops/
    sec
ML-DSA 65        verify  11600 ops took 1.005 sec, avg 0.087 ms, 11544.122 ops
    /sec
ML-DSA 87        key gen 7700 ops took 1.013 sec, avg 0.132 ms, 7598.278 ops/
    sec
ML-DSA 87        sign    2600 ops took 1.000 sec, avg 0.385 ms, 2599.634 ops/
    sec
ML-DSA 87        verify  7200 ops took 1.007 sec, avg 0.140 ms, 7152.274 ops/
    sec
```

I.5.4.2 Benchmarks from wolfSSL The following benchmarks were obtained with the following configuration flags:

```
./configure --enable-kyber \
            --enable-dilithium \
            --disable-shared \
            --enable-intelasm \
            --enable-aesni \
            --enable-sp \
            --enable-sp-math \
            --enable-sp-asm \
            CFLAGS="-Os -DECC_USER_CURVES -DHAVE_ECC256"
```

Note: Only two cores are used for these benchmarks.

```
wolfSSL Server Benchmark on TLS13-AES128-GCM-SHA256 with group ECC_SECP256R1:
```

```
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   965.511 ms
        Tx Total    :     7.469 ms
        Rx          :     2.978 MB/s
        Tx          :   384.903 MB/s
        Connect     :    48.343 ms
        Connect Avg :     2.014 ms
wolfSSL Client Benchmark on TLS13-AES128-GCM-SHA256 with group ECC_SECP256R1:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   967.748 ms
        Tx Total    :     6.789 ms
        Rx          :     2.971 MB/s
        Tx          :   423.496 MB/s
        Connect     :    48.574 ms
        Connect Avg :     2.024 ms

wolfSSL Server Benchmark on TLS13-AES128-GCM-SHA256 with group ECC_SECP384R1:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   960.296 ms
        Tx Total    :     7.494 ms
        Rx          :     2.994 MB/s
        Tx          :   383.617 MB/s
        Connect     :    56.255 ms
        Connect Avg :     2.344 ms
wolfSSL Client Benchmark on TLS13-AES128-GCM-SHA256 with group ECC_SECP384R1:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   962.002 ms
        Tx Total    :     7.367 ms
        Rx          :     2.989 MB/s
        Tx          :   390.259 MB/s
        Connect     :    56.220 ms
        Connect Avg :     2.343 ms

wolfSSL Server Benchmark on TLS13-AES128-GCM-SHA256 with group ECC_SECP521R1:
        Total       :   5767168 bytes
        Num Conns   :        23
        Rx Total    :   938.745 ms
        Tx Total    :     7.889 ms
        Rx          :     2.929 MB/s
        Tx          :   348.596 MB/s
        Connect     :    61.261 ms
        Connect Avg :     2.664 ms
wolfSSL Client Benchmark on TLS13-AES128-GCM-SHA256 with group ECC_SECP521R1:
        Total       :   5767168 bytes
        Num Conns   :        23
        Rx Total    :   940.382 ms
        Tx Total    :     7.540 ms
        Rx          :     2.924 MB/s
        Tx          :   364.711 MB/s
        Connect     :    61.433 ms
```

```
        Connect Avg :     2.671 ms

wolfSSL Server Benchmark on TLS13-AES128-GCM-SHA256 with group ML_KEM_512:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   952.389 ms
        Tx Total    :     5.561 ms
        Rx          :     3.019 MB/s
        Tx          :   517.005 MB/s
        Connect     :    50.177 ms
        Connect Avg :     2.091 ms
wolfSSL Client Benchmark on TLS13-AES128-GCM-SHA256 with group ML_KEM_512:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   954.202 ms
        Tx Total    :     4.751 ms
        Rx          :     3.013 MB/s
        Tx          :   605.110 MB/s
        Connect     :    48.602 ms
        Connect Avg :     2.025 ms

wolfSSL Server Benchmark on TLS13-AES128-GCM-SHA256 with group ML_KEM_768:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   955.030 ms
        Tx Total    :     5.882 ms
        Rx          :     3.010 MB/s
        Tx          :   488.757 MB/s
        Connect     :    51.283 ms
        Connect Avg :     2.137 ms
wolfSSL Client Benchmark on TLS13-AES128-GCM-SHA256 with group ML_KEM_768:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   955.658 ms
        Tx Total    :     6.200 ms
        Rx          :     3.008 MB/s
        Tx          :   463.686 MB/s
        Connect     :    49.717 ms
        Connect Avg :     2.072 ms

wolfSSL Server Benchmark on TLS13-AES128-GCM-SHA256 with group ML_KEM_1024:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   973.042 ms
        Tx Total    :     7.294 ms
        Rx          :     2.955 MB/s
        Tx          :   394.150 MB/s
        Connect     :    51.750 ms
        Connect Avg :     2.156 ms
wolfSSL Client Benchmark on TLS13-AES128-GCM-SHA256 with group ML_KEM_1024:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   973.655 ms
        Tx Total    :     7.996 ms
```

```
        Rx          :     2.953 MB/s
        Tx          :   359.573 MB/s
        Connect     :    50.328 ms
        Connect Avg :     2.097 ms

wolfSSL Server Benchmark on TLS13-AES128-GCM-SHA256 with group
   SecP256r1MLKEM512:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   961.483 ms
        Tx Total    :     7.430 ms
        Rx          :     2.990 MB/s
        Tx          :   386.966 MB/s
        Connect     :    55.885 ms
        Connect Avg :     2.329 ms
wolfSSL Client Benchmark on TLS13-AES128-GCM-SHA256 with group
   SecP256r1MLKEM512:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   963.042 ms
        Tx Total    :     7.088 ms
        Rx          :     2.985 MB/s
        Tx          :   405.605 MB/s
        Connect     :    53.236 ms
        Connect Avg :     2.218 ms

wolfSSL Server Benchmark on TLS13-AES128-GCM-SHA256 with group
   SecP384r1MLKEM768:
        Total       :   5767168 bytes
        Num Conns   :        23
        Rx Total    :   927.519 ms
        Tx Total    :     7.338 ms
        Rx          :     2.965 MB/s
        Tx          :   374.747 MB/s
        Connect     :    64.464 ms
        Connect Avg :     2.803 ms
wolfSSL Client Benchmark on TLS13-AES128-GCM-SHA256 with group
   SecP384r1MLKEM768:
        Total       :   5767168 bytes
        Num Conns   :        23
        Rx Total    :   929.281 ms
        Tx Total    :     6.923 ms
        Rx          :     2.959 MB/s
        Tx          :   397.229 MB/s
        Connect     :    60.200 ms
        Connect Avg :     2.617 ms

wolfSSL Server Benchmark on TLS13-AES128-GCM-SHA256 with group
   SecP521r1MLKEM1024:
        Total       :   5767168 bytes
        Num Conns   :        23
        Rx Total    :   918.122 ms
        Tx Total    :     7.598 ms
        Rx          :     2.995 MB/s
```

```
        Tx          :   361.941 MB/s
        Connect     :    79.426 ms
        Connect Avg :     3.453 ms
wolfSSL Client Benchmark on TLS13-AES128-GCM-SHA256 with group
   SecP521r1MLKEM1024:
        Total       :   5767168 bytes
        Num Conns   :        23
        Rx Total    :   919.900 ms
        Tx Total    :     7.563 ms
        Rx          :     2.989 MB/s
        Tx          :   363.618 MB/s
        Connect     :    71.686 ms
        Connect Avg :     3.117 ms

wolfSSL Server Benchmark on TLS13-AES256-GCM-SHA384 with group ECC_SECP256R1:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   962.723 ms
        Tx Total    :     6.394 ms
        Rx          :     2.986 MB/s
        Tx          :   449.663 MB/s
        Connect     :    52.042 ms
        Connect Avg :     2.168 ms
wolfSSL Client Benchmark on TLS13-AES256-GCM-SHA384 with group ECC_SECP256R1:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   963.166 ms
        Tx Total    :     7.537 ms
        Rx          :     2.985 MB/s
        Tx          :   381.433 MB/s
        Connect     :    52.348 ms
        Connect Avg :     2.181 ms



wolfSSL Server Benchmark on TLS13-AES256-GCM-SHA384 with group ECC_SECP384R1:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   966.071 ms
        Tx Total    :     8.458 ms
        Rx          :     2.976 MB/s
        Tx          :   339.929 MB/s
        Connect     :    56.135 ms
        Connect Avg :     2.339 ms
wolfSSL Client Benchmark on TLS13-AES256-GCM-SHA384 with group ECC_SECP384R1:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   968.053 ms
        Tx Total    :     7.895 ms
        Rx          :     2.970 MB/s
        Tx          :   364.155 MB/s
        Connect     :    56.188 ms
        Connect Avg :     2.341 ms
```

```
wolfSSL Server Benchmark on TLS13-AES256-GCM-SHA384 with group ECC_SECP521R1:
        Total       :   5767168 bytes
        Num Conns   :        23
        Rx Total    :   930.195 ms
        Tx Total    :     7.849 ms
        Rx          :     2.956 MB/s
        Tx          :   350.364 MB/s
        Connect     :    62.644 ms
        Connect Avg :     2.724 ms
wolfSSL Client Benchmark on TLS13-AES256-GCM-SHA384 with group ECC_SECP521R1:
        Total       :   5767168 bytes
        Num Conns   :        23
        Rx Total    :   932.128 ms
        Tx Total    :     7.440 ms
        Rx          :     2.950 MB/s
        Tx          :   369.619 MB/s
        Connect     :    62.538 ms
        Connect Avg :     2.719 ms

wolfSSL Server Benchmark on TLS13-AES256-GCM-SHA384 with group ML_KEM_512:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   973.208 ms
        Tx Total    :     8.190 ms
        Rx          :     2.954 MB/s
        Tx          :   351.021 MB/s
        Connect     :    49.608 ms
        Connect Avg :     2.067 ms
wolfSSL Client Benchmark on TLS13-AES256-GCM-SHA384 with group ML_KEM_512:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   975.874 ms
        Tx Total    :     7.051 ms
        Rx          :     2.946 MB/s
        Tx          :   407.772 MB/s
        Connect     :    48.708 ms
        Connect Avg :     2.030 ms

wolfSSL Server Benchmark on TLS13-AES256-GCM-SHA384 with group ML_KEM_768:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   965.259 ms
        Tx Total    :     8.098 ms
        Rx          :     2.978 MB/s
        Tx          :   355.041 MB/s
        Connect     :    51.284 ms
        Connect Avg :     2.137 ms
wolfSSL Client Benchmark on TLS13-AES256-GCM-SHA384 with group ML_KEM_768:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   967.507 ms
        Tx Total    :     7.774 ms
        Rx          :     2.972 MB/s
        Tx          :   369.828 MB/s
```

```
        Connect     :    49.899 ms
        Connect Avg :     2.079 ms

wolfSSL Server Benchmark on TLS13-AES256-GCM-SHA384 with group ML_KEM_1024:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   972.588 ms
        Tx Total    :     7.835 ms
        Rx          :     2.956 MB/s
        Tx          :   366.959 MB/s
        Connect     :    52.259 ms
        Connect Avg :     2.177 ms
wolfSSL Client Benchmark on TLS13-AES256-GCM-SHA384 with group ML_KEM_1024:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   974.238 ms
        Tx Total    :     7.838 ms
        Rx          :     2.951 MB/s
        Tx          :   366.813 MB/s
        Connect     :    50.758 ms
        Connect Avg :     2.115 ms

wolfSSL Server Benchmark on TLS13-AES256-GCM-SHA384 with group
   SecP256r1MLKEM512:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   971.832 ms
        Tx Total    :     7.544 ms
        Rx          :     2.958 MB/s
        Tx          :   381.096 MB/s
        Connect     :    54.727 ms
        Connect Avg :     2.280 ms
wolfSSL Client Benchmark on TLS13-AES256-GCM-SHA384 with group
   SecP256r1MLKEM512:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   972.623 ms
        Tx Total    :     8.807 ms
        Rx          :     2.956 MB/s
        Tx          :   326.456 MB/s
        Connect     :    52.613 ms
        Connect Avg :     2.192 ms

wolfSSL Server Benchmark on TLS13-AES256-GCM-SHA384 with group
   SecP384r1MLKEM768:
        Total       :   5767168 bytes
        Num Conns   :        23
        Rx Total    :   921.217 ms
        Tx Total    :     7.740 ms
        Rx          :     2.985 MB/s
        Tx          :   355.285 MB/s
        Connect     :    69.367 ms
        Connect Avg :     3.016 ms
wolfSSL Client Benchmark on TLS13-AES256-GCM-SHA384 with group
```

```
    SecP384r1MLKEM768:
         Total       :   5767168 bytes
         Num Conns   :        23
         Rx Total    :   923.622 ms
         Tx Total    :     6.928 ms
         Rx          :     2.977 MB/s
         Tx          :   396.956 MB/s
         Connect     :    63.739 ms
         Connect Avg :     2.771 ms

wolfSSL Server Benchmark on TLS13-AES256-GCM-SHA384 with group
    SecP521r1MLKEM1024:
         Total       :   5767168 bytes
         Num Conns   :        23
         Rx Total    :   920.447 ms
         Tx Total    :     7.735 ms
         Rx          :     2.988 MB/s
         Tx          :   355.548 MB/s
         Connect     :    78.446 ms
         Connect Avg :     3.411 ms
wolfSSL Client Benchmark on TLS13-AES256-GCM-SHA384 with group
    SecP521r1MLKEM1024:
         Total       :   5767168 bytes
         Num Conns   :        23
         Rx Total    :   921.889 ms
         Tx Total    :     7.585 ms
         Rx          :     2.983 MB/s
         Tx          :   362.578 MB/s
         Connect     :    71.310 ms
         Connect Avg :     3.100 ms
```

I.6 Documentation

Technical documentation and other resources such as known answer tests can be found at the NIST PQC website:

https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

For more algorithm-specific benchmarking information, the OQS Project has benchmarking information at their website:

https://openquantumsafe.org/benchmarking/

I.7 Post-Quantum Stateful Hash-Based Signatures

This section covers post-quantum stateful hash-based signature (HBS) schemes such as LMS/HSS, and XMSS/XMSS^MT, for which wolfSSL has recently added support.

I.7.1 Motivation

Stateful HBS schemes are of growing interest for a number of reasons. The primary motivation for stateful HBS schemes is post-quantum security. As discussed previously in this appendix, Shor's algorithm would allow a quantum computer to efficiently factorize large integers and compute discrete logarithms, thus completely breaking public-key cryptography schemes such as RSA and ECC.

In contrast, stateful HBS schemes are founded on the security of their underlying hash functions and Merkle trees (typically implemented with SHA256), which are not expected to be broken by the advent of cryptographically relevant quantum computers. For these reasons they have been recommended by NIST SP 800-208 and the NSA's CNSA 2.0 suite. See these two links for more info:

- https://csrc.nist.gov/publications/detail/sp/800-208/final
- https://media.defense.gov/2022/Sep/07/2003071834/-1/-1/0/CSA_CNSA_2.0_ALGORITHMS_.PDF

Furthermore, the CNSA 2.0 timeline has specified that post-quantum stateful HBS schemes should be used exclusively by 2030, and adoption should begin *immediately*. In fact, adoption of LMS is the earliest requirement in the CNSA 2.0 suite timeline.

However, the nature of stateful HBS schemes requires that significant care is given to their use and tracking their state. In a stateful HBS system, the private key is actually a finite set of one-time signature (OTS) keys, which may never be reused. If the same OTS key were used to sign two different messages, it would be possible for an attacker to fabricate signatures, and the security of the entire scheme would unravel. Therefore, stateful HBS schemes are not suitable for general use such as the public internet.

Instead, because of these unique strengths and characteristics, and NIST and NSA backing, stateful HBS schemes such as LMS/HSS are of particular interest for offline firmware authentication and signature verification, especially on embedded or constrained systems that are expected to have a long operational lifetime and thus need to be resilient against a cryptographically relevant quantum computer.

I.7.2 LMS/HSS signatures

wolfSSL is adding support for the LMS/HSS hash-based signature scheme to our wolfCrypt embedded crypto engine. This will be achieved by experimental integration with the hash-sigs LMS/HSS library (https://github.com/cisco/hash-sigs), similar to our previous libOQS integration.

Leighton-Micali Signatures (LMS), and its multi-tree variant, the Hierarchical Signature System (HSS), is a post-quantum, stateful hash-based signature scheme. It is noted for having small public and private keys, and fast signing and verifying. Its signature sizes are larger, but are tunable via its Winternitz parameter. See these two links from RFC8554 for more details:

- LMS: https://datatracker.ietf.org/doc/html/rfc8554
- HSS: https://datatracker.ietf.org/doc/html/rfc8554#section-6

As previously discussed, the LMS/HSS signature system consists of a finite number of one-time signature (OTS) keys, and thus may only safely generate a finite number of signatures. However the number of signatures, and the signature size, are tunable via a set of defined parameters, which will be discussed next.

I.7.2.1 Supported Parameters LMS/HSS signatures are defined by 3 parameters: - levels: number of levels of Merkle trees. - height: height of an individual Merkle tree. - Winternitz: number of bits from hash used in a Winternitz chain. Used as a space-time tradeoff for the signature size.

wolfSSL supports all LMS/HSS parameters defined in RFC8554:

- levels = {1..8}
- height = {5, 10, 15, 20, 25}
- Winternitz = {1, 2, 4, 8}

The number of available signatures is: - N = 2 ** (levels * height)

For convenience some parameter sets have been predefined in the enum `wc_LmsParm`. Its values are shown in the table below:

COPYRIGHT ©2024 wolfSSL Inc.

parameter set	description
WC_LMS_PARM_NONE	Not set, use default (WC_LMS_PARM_L1_H15_W2)
WC_LMS_PARM_L1_H15_W2	1 level Merkle tree of 15 height, Winternitz 2
WC_LMS_PARM_L1_H15_W4	same as above, Winternitz 4
WC_LMS_PARM_L2_H10_W2	2 level Merkle tree of 10 height, Winternitz 4
WC_LMS_PARM_L2_H10_W4	same as above, Winternitz 4
WC_LMS_PARM_L2_H10_W8	same as above, Winternitz 8
WC_LMS_PARM_L3_H5_W2	3 level Merkle tree of 5 height, Winternitz 2
WC_LMS_PARM_L3_H5_W4	same as above, Winternitz 4
WC_LMS_PARM_L3_H5_W8	same as above, Winternitz 8
WC_LMS_PARM_L3_H10_W4	3 level Merkle tree of 10 height, Winternitz 4
WC_LMS_PARM_L4_H5_W8	4 level Merkle tree of 5 height, Winternitz 8

The signature size and number of signatures is shown with respect to the parameter set here:

parameter set	signature size	number of signatures
WC_LMS_PARM_L1_H15_W2	4784	32768
WC_LMS_PARM_L1_H15_W4	2672	32768
WC_LMS_PARM_L2_H10_W2	9300	1048576
WC_LMS_PARM_L2_H10_W4	5076	1048576
WC_LMS_PARM_L2_H10_W8	2964	1048576
WC_LMS_PARM_L3_H5_W2	13496	32768
WC_LMS_PARM_L3_H5_W4	7160	32768
WC_LMS_PARM_L3_H5_W8	3992	32768
WC_LMS_PARM_L3_H10_W4	7640	1073741824
WC_LMS_PARM_L4_H5_W8	5340	1048576

As can be seen from the tables, signature sizes are primarily determined by the levels and Winternitz parameters, and height to a lesser extent: - Larger levels values increase signature size significantly. - Larger height values increase signature size modestly. - Larger winternitz values will reduce the signature size, at the expense of longer key generation and sign/verify times.

Key generation time is strongly determined by the height of the first level tree. A 3 level, 5 height tree is much faster than 1 level, 15 height at initial key gen, even if the number of available signatures is the same.

I.7.2.2 LMS/HSS Build Instructions Please see the wolfSSL repo's INSTALL file (https://github.com/wolfSSL/wolfssl/ Item 17 (Building with hash-sigs lib for LMS/HSS support [EXPERIMENTAL]) has instructions on how to configure and build wolfSSL and the hash-sigs LMS/HSS library.

I.7.2.3 Benchmark Data The following benchmark data was taken on an 8-core Intel i7-8700 CPU @ 3.20GHz, on Fedora 38 (`6.2.9-300.fc38.x86_64`). The multi-threaded example used 4 worker threads and 4 cores, while the single-threaded example used only a single core.

As discussed in item 17 of the INSTALL file, the hash-sigs lib offers two static libraries: - `hss_lib.a`: a single-threaded version. - `hss_lib_thread.a`: a multi-threaded version.

The multi-threaded version will spawn worker threads to accelerate cpu intensive tasks, such as key generation. This will mainly speedup key generation and signing for all parameter values, and to a lesser extent will speedup verifying for larger levels values.

For reference, wolfSSL was built with the following to obtain both benchmarks:

```
  ./configure \
    --enable-static \
    --disable-shared \
    --enable-lms=yes \
    --with-liblms=<path to hash sigs install>
```

multi-threaded benchmark

The following is benchmark data obtained when built against the multi-threaded `hss_lib_thread.a`, which used 4 worker threads to parallelize intensive tasks, and used 4 cores.

```
./wolfcrypt/benchmark/benchmark -lms_hss
------------------------------------------------------------------------------
 wolfSSL version 5.6.3
------------------------------------------------------------------------------
Math:   Multi-Precision: Wolf(SP) word-size=64 bits=4096 sp_int.c
wolfCrypt Benchmark (block bytes 1048576, min 1.0 sec each)
LMS/HSS L2_H10_W2  9300     sign      1500 ops took 1.075 sec, avg 0.717 ms,
   1394.969 ops/sec
LMS/HSS L2_H10_W2  9300   verify      5200 ops took 1.002 sec, avg 0.193 ms,
   5189.238 ops/sec
LMS/HSS L2_H10_W4  5076     sign       800 ops took 1.012 sec, avg 1.265 ms,
   790.776 ops/sec
LMS/HSS L2_H10_W4  5076   verify      2500 ops took 1.003 sec, avg 0.401 ms,
   2493.584 ops/sec
LMS/HSS L3_H5_W4  7160     sign      1500 ops took 1.051 sec, avg 0.701 ms,
   1427.485 ops/sec
LMS/HSS L3_H5_W4  7160   verify      2700 ops took 1.024 sec, avg 0.379 ms,
   2636.899 ops/sec
LMS/HSS L3_H5_W8  3992     sign       300 ops took 1.363 sec, avg 4.545 ms,
   220.030 ops/sec
LMS/HSS L3_H5_W8  3992   verify       400 ops took 1.066 sec, avg 2.664 ms,
   375.335 ops/sec
LMS/HSS L3_H10_W4  7640     sign       900 ops took 1.090 sec, avg 1.211 ms,
   825.985 ops/sec
LMS/HSS L3_H10_W4  7640   verify      2400 ops took 1.037 sec, avg 0.432 ms,
   2314.464 ops/sec
LMS/HSS L4_H5_W8  5340     sign       300 ops took 1.310 sec, avg 4.367 ms,
   228.965 ops/sec
LMS/HSS L4_H5_W8  5340   verify       400 ops took 1.221 sec, avg 3.053 ms,
   327.599 ops/sec
Benchmark complete
```

single-threaded benchmark

The following is benchmark data obtained when built against the single-threaded `hss_lib.a`, which will use only a single core.

```
$ ./wolfcrypt/benchmark/benchmark -lms_hss
------------------------------------------------------------------------------
 wolfSSL version 5.6.3
------------------------------------------------------------------------------
Math:   Multi-Precision: Wolf(SP) word-size=64 bits=4096 sp_int.c
wolfCrypt Benchmark (block bytes 1048576, min 1.0 sec each)
LMS/HSS L2_H10_W2  9300     sign       800 ops took 1.115 sec, avg 1.394 ms,
   717.589 ops/sec
```

```
LMS/HSS L2_H10_W2  9300   verify      4500 ops took 1.001 sec, avg 0.223 ms,
   4493.623 ops/sec
LMS/HSS L2_H10_W4  5076     sign       500 ops took 1.239 sec, avg 2.478 ms,
   403.519 ops/sec
LMS/HSS L2_H10_W4  5076   verify      2100 ops took 1.006 sec, avg 0.479 ms,
   2087.944 ops/sec
LMS/HSS L3_H5_W4  7160     sign       800 ops took 1.079 sec, avg 1.349 ms,
   741.523 ops/sec
LMS/HSS L3_H5_W4  7160   verify      1600 ops took 1.012 sec, avg 0.632 ms,
   1581.686 ops/sec
LMS/HSS L3_H5_W8  3992     sign       100 ops took 1.042 sec, avg 10.420 ms,
   95.971 ops/sec
LMS/HSS L3_H5_W8  3992   verify       200 ops took 1.220 sec, avg 6.102 ms,
   163.894 ops/sec
LMS/HSS L3_H10_W4  7640     sign       400 ops took 1.010 sec, avg 2.526 ms,
   395.864 ops/sec
LMS/HSS L3_H10_W4  7640   verify      1500 ops took 1.052 sec, avg 0.701 ms,
   1426.284 ops/sec
LMS/HSS L4_H5_W8  5340     sign       100 ops took 1.066 sec, avg 10.665 ms,
   93.768 ops/sec
LMS/HSS L4_H5_W8  5340   verify       200 ops took 1.478 sec, avg 7.388 ms,
   135.358 ops/sec
Benchmark complete
```

I.7.3 XMSS/XMSS^MT signatures

wolfSSL is adding support for XMSS/XMSS^MT stateful hash-based signatures. Similar to LMS, this will be done by experimental integration with the xmss-reference repository (https://github.com/XMSS/xmss-reference.git), from RFC 8391 (https://www.rfc-editor.org/rfc/rfc8391.html).

The xmss-reference supports `xmss_core_fast`, and `xmss_core` implementations. The `xmss_core_fast` implementation was designed to prioritize performance with larger private key sizes as a tradeoff. Our integration uses `xmss_core_fast`, with a patch applied so that the wolfCrypt SHA256 implementation may be used instead.

The patch may be found here

```
in the wolfssl-examples repository: https://github.com/wolfSSL/wolfssl-
   examples.

Overall, XMSS/XMSS^MT is similar to LMS/HSS. For a more detailed comparison
   see
"LMS vs XMSS: Comparison of two Hash-Based Signature Standards" (https://
   eprint.iacr.org/2017/349.pdf).


XMSS^MT is the Multi-Tree generalization of XMSS,
analogous to HSS with LMS, with the distinction that the Winternitz value is
   fixed to `w=16` in
XMSS/XMSS^MT. The public key is slightly larger in XMSS/XMSS^MT (at 68 bytes
   in XMSS/XMSS^MT, vs 60 bytes in LMS/HSS), while signatures are slightly
   smaller.

#### Supported Parameters
```

wolfSSL supports the SHA256 XMSS/XMSS^MT parameter sets from Tables 10 and 11 from NIST SP 800-208 (https://csrc.nist.gov/pubs/sp/800/208/final).

parameter set name	Oid	n	w	h	d	h/d	Sig len
XMSS							
"XMSS-SHA2_10_256"	0x00000001	32	16	10	1	10	2500
"XMSS-SHA2_16_256"	0x00000002	32	16	16	1	16	2692
"XMSS-SHA2_20_256"	0x00000003	32	16	20	1	20	2820
XMSS^MT							
"XMSSMT-SHA2_20/2_256"	0x00000001	32	16	20	2	10	4963
"XMSSMT-SHA2_20/4_256"	0x00000002	32	16	20	4	5	9251
"XMSSMT-SHA2_40/2_256"	0x00000003	32	16	40	2	20	5605
"XMSSMT-SHA2_40/4_256"	0x00000004	32	16	40	4	10	9893
"XMSSMT-SHA2_40/8_256"	0x00000005	32	16	40	8	5	18469
"XMSSMT-SHA2_60/3_256"	0x00000006	32	16	60	3	20	8392
"XMSSMT-SHA2_60/6_256"	0x00000007	32	16	60	6	10	14824
"XMSSMT-SHA2_60/12_256"	0x00000008	32	16	60	12	5	27688

In the table above, `n` is the number of bytes in the HASH function, `w` the Winternitz value, `h` the total height of the tree system, and `d` the number of levels of trees.

Key generation time is strongly determined by the height of the
first level tree (or `h/d`), while signature length grows primarily with `d` (the number of hyper
tree levels).

Similar to LMS/HSS, the number of available signatures grows as `2**h`, where h is the
total height of the tree system.

Benchmark Data

In the following, benchmark data is shown for several XMSS/XMSS^MT parameter sets,
for intel x86_64 and aarch64. The SHA256 performance on these systems is also listed for reference,
as computing the large number of required hash chains will constitute the bulk of the CPU
work for XMSS/XMSS^MT. Additionally, our patch to xmss-reference substitutes wolfCrypt's SHA256 implementation, and therefore benefits from the same ASM speedups.

As previously mentioned, our xmss integration is using the `xmss_core_fast` implementation from xmss-reference, which has faster performance at the tradeoff of larger private key sizes.

x86_64

The following x86_64 benchmark data were taken on an 8-core Intel i7-8700 CPU @ 3.20GHz, on Fedora 38 (`6.2.9-300.fc38.x86_64`). This CPU has `avx avx2` flags, which can accelerate hash operations and be utilized with `--enable-intelasm`.

COPYRIGHT ©2024 wolfSSL Inc.

With `--enable-intelasm`:

```text
$./wolfcrypt/benchmark/benchmark -xmss_xmssmt -sha256
------------------------------------------------------------------------------
 wolfSSL version 5.6.3
------------------------------------------------------------------------------
Math:   Multi-Precision: Wolf(SP) word-size=64 bits=4096 sp_int.c
wolfCrypt Benchmark (block bytes 1048576, min 1.0 sec each)
SHA-256                     500 MiB took 1.009 seconds,  495.569 MiB/s Cycles
   per byte =   6.14
XMSS-SHA2_10_256  2500     sign       200 ops took 1.010 sec, avg 5.052 ms,
   197.925 ops/sec
XMSS-SHA2_10_256  2500   verify      1600 ops took 1.011 sec, avg 0.632 ms,
   1582.844 ops/sec
XMSSMT-SHA2_20/2_256  4963     sign       200 ops took 1.286 sec, avg 6.431 ms
   , 155.504 ops/sec
XMSSMT-SHA2_20/2_256  4963   verify       700 ops took 1.009 sec, avg 1.441 ms
   , 693.905 ops/sec
XMSSMT-SHA2_20/4_256  9251     sign       300 ops took 1.223 sec, avg 4.076 ms
   , 245.335 ops/sec
XMSSMT-SHA2_20/4_256  9251   verify       400 ops took 1.027 sec, avg 2.569 ms
   , 389.329 ops/sec
XMSSMT-SHA2_40/4_256  9893     sign       200 ops took 1.466 sec, avg 7.332 ms
   , 136.394 ops/sec
XMSSMT-SHA2_40/4_256  9893   verify       400 ops took 1.024 sec, avg 2.560 ms
   , 390.627 ops/sec
XMSSMT-SHA2_40/8_256 18469     sign       300 ops took 1.202 sec, avg 4.006 ms
   , 249.637 ops/sec
XMSSMT-SHA2_40/8_256 18469   verify       200 ops took 1.089 sec, avg 5.446 ms
   , 183.635 ops/sec
XMSSMT-SHA2_60/6_256 14824     sign       200 ops took 1.724 sec, avg 8.618 ms
   , 116.033 ops/sec
XMSSMT-SHA2_60/6_256 14824   verify       300 ops took 1.136 sec, avg 3.788 ms
   , 263.995 ops/sec
XMSSMT-SHA2_60/12_256 27688     sign       300 ops took 1.210 sec, avg 4.034
   ms, 247.889 ops/sec
XMSSMT-SHA2_60/12_256 27688   verify       200 ops took 1.575 sec, avg 7.877
   ms, 126.946 ops/sec
Benchmark complete
```

Without `--enable-intelasm`:

```
$./wolfcrypt/benchmark/benchmark -xmss_xmssmt -sha256
------------------------------------------------------------------------------
 wolfSSL version 5.6.3
------------------------------------------------------------------------------
Math:   Multi-Precision: Wolf(SP) word-size=64 bits=4096 sp_int.c
wolfCrypt Benchmark (block bytes 1048576, min 1.0 sec each)
SHA-256                     275 MiB took 1.005 seconds,  273.549 MiB/s Cycles
   per byte =  11.13
XMSS-SHA2_10_256  2500     sign       200 ops took 1.356 sec, avg 6.781 ms,
   147.480 ops/sec
XMSS-SHA2_10_256  2500   verify      1200 ops took 1.025 sec, avg 0.854 ms,
```

```
   1170.547 ops/sec
XMSSMT-SHA2_20/2_256  4963     sign       200 ops took 1.687 sec, avg 8.436 ms
   , 118.546 ops/sec
XMSSMT-SHA2_20/2_256  4963   verify       600 ops took 1.187 sec, avg 1.978 ms
   , 505.663 ops/sec
XMSSMT-SHA2_20/4_256  9251     sign       200 ops took 1.119 sec, avg 5.593 ms
   , 178.785 ops/sec
XMSSMT-SHA2_20/4_256  9251   verify       300 ops took 1.086 sec, avg 3.622 ms
   , 276.122 ops/sec
XMSSMT-SHA2_40/4_256  9893     sign       200 ops took 1.991 sec, avg 9.954 ms
   , 100.460 ops/sec
XMSSMT-SHA2_40/4_256  9893   verify       300 ops took 1.043 sec, avg 3.478 ms
   , 287.545 ops/sec
XMSSMT-SHA2_40/8_256 18469     sign       200 ops took 1.114 sec, avg 5.572 ms
   , 179.454 ops/sec
XMSSMT-SHA2_40/8_256 18469   verify       200 ops took 1.495 sec, avg 7.476 ms
   , 133.770 ops/sec
XMSSMT-SHA2_60/6_256 14824     sign       100 ops took 1.111 sec, avg 11.114
   ms, 89.975 ops/sec
XMSSMT-SHA2_60/6_256 14824   verify       200 ops took 1.070 sec, avg 5.349 ms
   , 186.963 ops/sec
XMSSMT-SHA2_60/12_256 27688     sign       200 ops took 1.148 sec, avg 5.739
   ms, 174.247 ops/sec
XMSSMT-SHA2_60/12_256 27688   verify       100 ops took 1.080 sec, avg 10.797
   ms, 92.618 ops/sec
Benchmark complete
```

aarch64

The following aarch64 data were taken on Ubuntu linux (`5.15.0-71-generic`) running on an Apple M1, with cpu flags `sha1 sha2 sha3 sha512`, which will specifically significantly accelerate SHA hash operations when built with `--enable-armasm`.

With `--enable-armasm`:

```
$ ./wolfcrypt/benchmark/benchmark -xmss_xmssmt -sha256
------------------------------------------------------------------------------
 wolfSSL version 5.6.3
------------------------------------------------------------------------------
Math: Multi-Precision: Wolf(SP) word-size=64 bits=4096 sp_int.c
wolfCrypt Benchmark (block bytes 1048576, min 1.0 sec each)
SHA-256                  2305 MiB took 1.001 seconds, 2303.346 MiB/s
XMSS-SHA2_10_256  2500     sign       800 ops took 1.079 sec, avg 1.349 ms,
   741.447 ops/sec
XMSS-SHA2_10_256  2500   verify      6500 ops took 1.007 sec, avg 0.155 ms,
   6455.445 ops/sec
XMSSMT-SHA2_20/2_256  4963     sign       700 ops took 1.155 sec, avg 1.650 ms
   , 606.154 ops/sec
XMSSMT-SHA2_20/2_256  4963   verify      3100 ops took 1.021 sec, avg 0.329 ms
   , 3037.051 ops/sec
XMSSMT-SHA2_20/4_256  9251     sign      1100 ops took 1.006 sec, avg 0.915 ms
   , 1093.191 ops/sec
XMSSMT-SHA2_20/4_256  9251   verify      1700 ops took 1.013 sec, avg 0.596 ms
   , 1677.399 ops/sec
XMSSMT-SHA2_40/4_256  9893     sign       600 ops took 1.096 sec, avg 1.827 ms
   , 547.226 ops/sec
```

```
XMSSMT-SHA2_40/4_256  9893   verify      1600 ops took 1.062 sec, avg 0.664 ms
    , 1506.946 ops/sec
XMSSMT-SHA2_40/8_256 18469     sign      1100 ops took 1.007 sec, avg 0.916 ms
    , 1092.214 ops/sec
XMSSMT-SHA2_40/8_256 18469   verify       900 ops took 1.088 sec, avg 1.209 ms
    , 827.090 ops/sec
XMSSMT-SHA2_60/6_256 14824     sign       600 ops took 1.179 sec, avg 1.966 ms
    , 508.728 ops/sec
XMSSMT-SHA2_60/6_256 14824   verify      1100 ops took 1.038 sec, avg 0.944 ms
    , 1059.590 ops/sec
XMSSMT-SHA2_60/12_256 27688     sign      1100 ops took 1.015 sec, avg 0.923
    ms, 1083.767 ops/sec
XMSSMT-SHA2_60/12_256 27688   verify       600 ops took 1.149 sec, avg 1.914
    ms, 522.367 ops/sec
Benchmark complete
```

Without `--enable-armasm`:

```
$ ./wolfcrypt/benchmark/benchmark -xmss_xmssmt -sha256
------------------------------------------------------------------------------
 wolfSSL version 5.6.3
------------------------------------------------------------------------------
Math: Multi-Precision: Wolf(SP) word-size=64 bits=4096 sp_int.c
wolfCrypt Benchmark (block bytes 1048576, min 1.0 sec each)
SHA-256                     190 MiB took 1.020 seconds,  186.277 MiB/s
XMSS-SHA2_10_256  2500     sign       200 ops took 1.908 sec, avg 9.538 ms,
    104.845 ops/sec
XMSS-SHA2_10_256  2500   verify       800 ops took 1.002 sec, avg 1.253 ms,
    798.338 ops/sec
XMSSMT-SHA2_20/2_256  4963     sign       100 ops took 1.084 sec, avg 10.843
    ms, 92.222 ops/sec
XMSSMT-SHA2_20/2_256  4963   verify       500 ops took 1.240 sec, avg 2.479 ms
    , 403.334 ops/sec
XMSSMT-SHA2_20/4_256  9251     sign       200 ops took 1.615 sec, avg 8.074 ms
    , 123.855 ops/sec
XMSSMT-SHA2_20/4_256  9251   verify       200 ops took 1.071 sec, avg 5.355 ms
    , 186.726 ops/sec
XMSSMT-SHA2_40/4_256  9893     sign       100 ops took 1.354 sec, avg 13.543
    ms, 73.840 ops/sec
XMSSMT-SHA2_40/4_256  9893   verify       300 ops took 1.483 sec, avg 4.945 ms
    , 202.237 ops/sec
XMSSMT-SHA2_40/8_256 18469     sign       200 ops took 1.588 sec, avg 7.941 ms
    , 125.922 ops/sec
XMSSMT-SHA2_40/8_256 18469   verify       100 ops took 1.042 sec, avg 10.415
    ms, 96.014 ops/sec
XMSSMT-SHA2_60/6_256 14824     sign       100 ops took 1.571 sec, avg 15.710
    ms, 63.654 ops/sec
XMSSMT-SHA2_60/6_256 14824   verify       200 ops took 1.526 sec, avg 7.632 ms
    , 131.033 ops/sec
XMSSMT-SHA2_60/12_256 27688     sign       200 ops took 1.607 sec, avg 8.036
    ms, 124.434 ops/sec
XMSSMT-SHA2_60/12_256 27688   verify       100 ops took 1.501 sec, avg 15.011
    ms, 66.616 ops/sec
Benchmark complete
```

I.7.4 Developer Notes

- If you are trying to thwart the “harvest now, decrypt later” threat model and you are willing to sacrifice some interoperability, then you do not want to advertise support for conventional algorithms in the Supported Groups extension. Be sure to call `wolfSSL_UseKeyShare()` and `wolfSSL_set_groups()` with your chosen algorithms. Only calling `wolfSSL_UseKeyShare()` is insufficient as that will advertise your support for quantum-vulnerable algorithms. If your peer does not support post-quantum algorithms, they will then send a HelloRetryRequest which will then result in a connection with a conventional algorithm.

J wolfSSL Porting Guide

J.1 Purpose

This guide provides a reference for developers and engineers porting the wolfSSL lightweight SSL/TLS library to new embedded platforms, operating systems, or transport mediums (TCP/IP, bluetooth, etc.). It calls out areas in the wolfSSL codebase which typically require modification when porting wolfSSL. It should be considered a "guide" and as such, it is an evolving work. If there is something you find missing, please let us know and we'll be happy to add instructions or clarification to the document.

J.2 Audience

This guide caters to developers or engineers porting the wolfSSL and wolfCrypt to new platforms or environments that are not supported by default.

J.3 Introduction

Several steps need to be iterated through when getting wolfSSL to run on an embedded platform. Some of these steps are outlined in Section 2.4.

Apart from steps in Chapter 2 of the wolfSSL Manual, there are areas in the code which may need porting or modifications in order to accommodate a specific platform. wolfSSL abstracts many of these areas - attempting to make it as easy as possible to port wolfSSL to a new platform.

In the `./wolfssl/wolfcrypt/settings.h` file, there are several defines specific to different operating systems, TCP/IP stacks, and chipsets (ex: MBED, FREESCALE_MQX, MICROCHIP_PIC32, MICRIUM, EBSNET, etc.). There are two main locations to put `#defines` when compiling and porting wolfSSL to a new platform:

1. New defines for a Operating System or TCP/IP stack port are typically added to the `settings.h` file when a new port of wolfSSL is completed. This provides an easy way to turn on/off features as well as customize build settings that should be "default" for that build. New custom defines can be added in this file when doing a port of wolfSSL to a new platform. We encourage users to contribute ports of wolfSSL back to the master open source code branch on GitHub. This helps keep wolfSSL up to date and allows different ports to remain updated as the wolfSSL project improves and moves forward.
2. For users not wanting to contribute back their changes to wolfSSL proper, or for users who want to customize the wolfSSL build with additional preprocessor defines, wolfSSL recommends the use of a custom `user_settings.h` header file. If `WOLFSSL_USER_SETTINGS` is defined when compiling the wolfSSL source files, wolfSSL will automatically include a custom header file called `user_settings.h`. This header should be created by the user and placed on the include path. This allows users to maintain one single file for their wolfSSL build, and makes it much easier to update to newer versions of wolfSSL.

wolfSSL encourages the submission of patches and code changes through either direct email (facts@wolfssl.com), or through GitHub pull request.

J.4 Porting wolfSSL

J.4.1 Data Types

Q: When do I need to read this section?

A: Setting the correct data type size for your platform is always important.

wolfSSL benefits speed-wise from having a 64-bit type available. Define SIZEOF_LONG and SIZEOF_LONG_LONG to match the result of sizeof(long) and sizeof(long long) on your platform. This can be added to a custom define in the settings.h file or to user_settings.h. For example, in settings.h under a sample define of MY_NEW_PLATFORM:

```
#ifdef MY_NEW_PLATFORM
    #define SIZEOF_LONG 4
    #define SIZEOF_LONG_LONG 8
    ...
#endif
```

There are two additional data types used by wolfSSL and wolfCrypt, called word32 and word16. The default type mappings for these are:

```
#ifndef WOLFSSL_TYPES
#ifndef byte
    typedef unsigned char  byte;
#endif
    typedef unsigned short word16;
    typedef unsigned int   word32;
    typedef byte           word24[3];
#endif
```

word32 should be mapped to the compiler's 32-bit type, and word16 to the compiler's 16-bit type. If these default mappings are incorrect for your platform, you should define WOLFSSL_TYPES in settings.h or user_settings.h and assign your own custom typedefs for word32 and word16.

The fastmath library in wolfSSL uses the fp_digit and fp_word types. By default these are mapped in <wolfssl/wolfcrypt/tfm.h> depending on build configuration.

fp_word should be twice the size of fp_digit. If the default cases do not hold true for your platform, you should define WOLFSSL_BIGINT_TYPES in settings.h or user_settings.h and assign your own custom typedefs for fp_word and fp_digit.

wolfSSL does use a 64-bit type when available for some operations. The wolfSSL build tries to detect and set up the correct underlying data type for word64 based on what SIZEOF_LONG and SIZEOF_LONG_LONG have been set to. On some platforms that don't have a true 64-bit type, where two 32-bit types are used in conjunction, performance can be slow. To compile out the use of 64-bit types, define NO_64BIT.

J.4.2 Endianness

Q: When do I need to read this section?

A: Your platform is a big endian system.

Is your platform big endian or little endian? wolfSSL defaults to a little endian system. If your system is big endian, define BIG_ENDIAN_ORDER when building wolfSSL. Example of setting this in settings.h:

```
#ifdef MY_NEW_PLATFORM
    ...
    #define BIG_ENDIAN_ORDER
    ...
#endif
```

J.4.3 writev

Q: When do I need to read this section?

A: <sys/uio.h> is not available.

By default, the wolfSSL API makes available wolfSSL_writev() to applications, which simulates writev() semantics. On systems that don't have the <sys/uio.h> header available, define NO_WRITEV to exclude this feature.

J.4.4 Input / Output

Q: When do I need to read this section?

A: A BSD-style socket API is not available, you are using a custom transport layer or TCP/IP stack, or only want to use static buffers.

wolfSSL defaults to using a BSD-style socket interface. If your transport layer provides a BSD socket interface, wolfSSL should integrate into it as-is, unless custom headers are needed.

wolfSSL provides a custom I/O abstraction layer which allows users to tailor wolfSSL's I/O functionality to their system. Full details can be found in Section 5.1.2.

Simply put, you can define WOLFSSL_USER_IO, then write your own I/O callback functions using wolfSSL's default EmbedSend() and EmbedReceive() as templates. These two functions are located in ./src/io.c.

wolfSSL uses dynamic buffers for input and output, which default to 0 bytes. If an input record is received that is greater in size than the buffer, then a dynamic buffer is temporarily used to handle the request and then freed.

If you prefer using large, 16kB static buffers which will never need dynamic memory, you can enable this option by defining LARGE_STATIC_BUFFERS.

If dynamic buffers are used and the user requests an wolfSSL_write() that is bigger than the buffer size, then a dynamic block up to MAX_RECORD_SIZE is used to send the data. Users wishing to only send the data in chunks of the current buffer size at maximum, as defined by RECORD_SIZE, can do this by defining STATIC_CHUNKS_ONLY. When using this define, RECORD_SIZE defaults to 128 bytes.

J.4.5 Filesystem

Q: When do I need to read this section?

A: No file system is available, standard file system functions are not available, or you have a custom file system.

wolfSSL uses the filesystem for loading keys and certificates into the SSL session or context. wolfSSL also allows loading these from memory buffers. If strictly using memory buffers, a filesystem is not needed.

You can disable wolfSSL's usage of the filesystem by defining NO_FILESYSTEM when building the library. This means that certificates and keys will need to be loaded from memory buffers instead of files. An example of setting this in settings.h:

```
#ifdef MY_NEW_PLATFORM
    ...
    #define NO_FILESYSTEM
    ...
#endif
```

Test key and certificate buffers can be found in the ./wolfssl/certs_test.h header file. These will match up to corresponding certificates and keys found in the ./certs directory.

The certs_test.h header file can be updated using the ./gencertbuf.pl script if needed. Inside gencertbuf.pl, there are two arrays: fileList_1024 and fileList_2048. Additional certificates

or keys may be added to the respective array, depending on key size, and must be in DER format. The above mentioned arrays map a certificate/key file location with the desired buffer name. After modifying gencertbuf.pl, running it from the wolfSSL root directory will update the certificate and key buffers in ./wolfssl/certs_test.h:

```
./gencertbuf.pl
```

If you would like to use a filesystem other than the default, the filesystem abstraction layer is located in ./wolfssl/wolfcrypt/wc_port.h. Here you will see filesystem ports for various platforms including EBSNET, FREESCALE_MQX, and MICRIUM. You can add a custom define for your platform if needed - allowing you to define file system functions with XFILE, XFOPEN, XFSEEK, etc. For example, the filesystem layer in wc_port.h for Micrium's µC/OS (MICRIUM) is as follows:

```
#elif defined(MICRIUM)
#include <fs.h>
#define XFILE      FS_FILE*
#define XFOPEN     fs_fopen
#define XFSEEK     fs_fseek
#define XFTELL     fs_ftell
#define XREWIND    fs_rewind
#define XFREAD     fs_fread
#define XFCLOSE    fs_fclose
#define XSEEK_END  FS_SEEK_END
#define XBADFILE   NULL
```

J.4.6 Threading

Q: When do I need to read this section?

A: You want to use wolfSSL in a multithreaded environment, or want to just compile it in single threaded mode.

If wolfSSL will only be used in a single threaded environment, the wolfSSL mutex layer can be disabled when compiling wolfSSL by defining SINGLE_THREADED. This will negate the need to port the wolfSSL mutex layer.

If wolfSSL needs to be used in a multithreaded environment, the wolfSSL mutex layer will need to be ported to the new environment. The mutex layer can be found in ./wolfssl/wolfcrypt/wc_port.h and ./wolfcrypt/src/wc_port.c. wolfSSL_Mutex will need to be defined for the new system in wc_port.h and the mutex functions (wc_InitMutex, wc_FreeMutex, wc_LockMutex and wc_UnLockMutex) in wc_port.c. You can search in wc_port.h and wc_port.c to see an example for some existing platform port layers (EBSNET, FREESCALE_MQX, etc.).

J.4.7 Random Seed

Q: When do I need to read this section?

A: Either /dev/random or /dev/urandom is not available or you want to integrate into a hardware RNG.

By default, wolfSSL uses /dev/urandom or /dev/random to generate a RNG seed. The NO_DEV_RANDOM define can be used when building wolfSSL to disable the default GenerateSeed() function. If this is defined, you need to write a custom GenerateSeed() function in ./wolfcrypt/src/random.c, specific to your target platform. This allows you to seed wolfSSL's PRNG with a hardware-based random entropy source if available.

For examples of how GenerateSeed() needs to be written, reference wolfSSL's existing GenerateSeed() implementations in ./wolfcrypt/src/random.c.

J.4.8 Memory

Q: When do I need to read this section?

A: When you don't have standard memory functions available or are interested in memory usage differences between optional math libraries.

wolfSSL proper uses both `malloc()` and `free()` by default. When using the normal big integer math library, wolfCrypt will also use `realloc()`.

By default wolfSSL/wolfCrypt use the normal big integer math library, which uses quite a bit of dynamic memory. When building wolfSSL, the fastmath library can be enabled, which is both faster and uses no dynamic memory for crypto operations (all on the stack). By using fastmath, wolfSSL won't need a `realloc()` implementation at all. As the SSL layer of wolfSSL still uses some dynamic memory, `malloc()` and `free()` are still required.

For a comparison of resource usage (stack/heap) between the big integer math library and fastmath library, ask us to see our Resource Use document.

To enable fastmath, define `USE_FAST_MATH` and build in `./wolfcrypt/src/tfm.c` instead of `./wolfcrypt/src/integer.c`. Since the stack memory can be large when using fastmath, we recommend defining `TFM_TIMING_RESISTANT` as well.

If the normal `malloc()`, `free()`, and possibly `realloc()` functions are not available, define `XMALLOC_USER`, then provide custom memory function hooks in `./wolfssl/wolfcrypt/types.h` specific to the target environment.

Please read section 5.1.1.1 for details about using `XMALLOC_USER`.

J.4.9 Time

Q: When do I need to read this section?

A: When standard time functions (`time()`, `gmtime()`) are not available, or you need to specify a custom clock tick function.

By default, wolfSSL uses `time()`, `gmtime()`, and `ValidateDate()`, as specified in `./wolfcrypt/src/asn.c`. These are abstracted to `XTIME`, `XGMTIME`, and `XVALIDATE_DATE`. If the standard time functions, and `time.h`, are not available, the user can define `USER_TIME`. After defining `USER_TIME`, the user can define their own `XTIME`, `XGMTIME`, and `XVALIDATE_DATE` functions.

wolfSSL uses `time(0)` by default for the clock tick function. This is located in `./src/internal.c` inside of the `LowResTimer()` function.

Defining `USER_TICKS` allows the user to define their own clock tick function if `time(0)` is not wanted. The custom function needs second accuracy, but doesn't have to be correlated to EPOCH. See `LowResTimer()` function in `./src/internal.c` for reference.

J.4.10 C Standard Library

Q: When do I need to read this section?

A: When you don't have a C standard library available, or have a custom one.

wolfSSL can be built without the C standard library to provide a higher level of portability and flexibility to developers. When doing so, the user needs to map functions they wish to use instead of the C standard ones.

Section 7, above, covered memory functions. In addition to memory function abstraction, wolfSSL also abstracts string function and math functions, where the specific functions are typically abstracted to a define in the form of X<FUNC>, where <FUNC> is the name of the function being abstracted.

Please read Section 5.1 for details.

J.4.11 Logging

Q: When do I need to read this section?

A: You want to enable debug messages but don't have stderr available.

By default, wolfSSL provides debug output through stderr. In order for debug messages to be enabled, wolfSSL must be compiled with `DEBUG_WOLFSSL` defined, and `wolfSSL_Debugging_ON()` must be called from the application code. `wolfSSL_Debugging_OFF()` may be used by the application layer to turn off wolfSSL debug messages.

For environments which do not have stderr available, or wish to output debug messages over a different output stream or in a different format, wolfSSL allows applications to register a logging callback.

Please read Section 8.1 for details.

J.4.12 Public Key Operations

Q: When do I need to read this section?

A: You want to use your own public key implementation with wolfSSL.

wolfSSL allows users to write their own public key callbacks which will be called when the SSL/TLS layer needs to do public key operations. The user can optionally define 6 functions:

1. ECC sign callback
2. ECC verify callback
3. RSA sign callback
4. RSA verify callback
5. RSA encrypt callback
6. RSA decrypt callback

For full details, please read Section 6.4.

J.4.13 Atomic Record Layer Processing

Q: When do I need to read this section?

A: You want to do your own processing of record layers, specifically MAC/encrypt and decrypt/verify operations.

By default, wolfSSL handles record layer processing for the user using its cryptography library, wolfCrypt. wolfSSL provides Atomic Record Processing callbacks for users who wish to have more control over MAC/encrypt and decrypt/verify functionality during the SSL/TLS connection.

The user will need to define 2 functions:

1. MAC/encrypt callback function
2. Decrypt/verify callback function

For full details, please read Section 6.3.

J.4.14 Features

Q: When do I need to read this section?

A: When you want to disable features.

Features can be disabled when building wolfSSL by using the appropriate defines. For a list of defines available, please refer to Chapter 2.

J.5 Next Steps

J.5.1 wolfCrypt Test Application

After getting wolfSSL proper to build on the target platform, a good next step is to port the wolfCrypt test application. Running this application on the target system will verify that all the crypto algorithms are working correctly, using NIST test vectors.

If this step is skipped, and you instead proceed directly to establishing an SSL connection, it can be more difficult to debug problems caused by underlying crypto operations failing.

The wolfCrypt test application is located in `./wolfcrypt/test/test.c`. If an embedded application has its own `main()` function, then `NO_MAIN_DRIVER` must be defined when compiling `./wolfcrypt/test/test.c`. This will allow the application's `main()` to call each cipher/algorithm test individually on its own.

If an embedded device does not have enough resources to run the entire wolfCrypt test application, individual tests can be broken out of `test.c` and compiled individually. Please ensure that correct header files needed for the specific test case are included in the build when extracting isolated crypto tests from `test.c`.

J.6 Support

General support questions may be sent directly to wolfSSL either through email, support forums, or wolfSSL's Zendesk ticket tracking system.

Website: https://www.wolfssl.com

Support Email: support@wolfssl.com

Zendesk: https://wolfssl.zendesk.com

Forums: https://www.wolfssl.com/forums

wolfSSL offers several support packages as well as consulting services to help users and customers port wolfSSL to new environments.

Support Packages: https://www.wolfssl.com/wolfSSL/Support/support_tiers.php

Consulting Services: https://www.wolfssl.com/wolfSSL/wolfssl-consulting.html

General Inquiries: facts@wolfssl.com

K wolfSM (ShangMi)

This appendix provides information about the Chinese National Standard's cryptographic algorithms known as ShangMi (SM) in wolfSSL.

wolfSM support includes: * SM3 - Hash Function * SM4 - Cipher * SM2 - ECDH key agreement and a signature scheme using the specified 256-bit elliptic curve.

The code must be installed into wolfSSL in order to be used.

Note that the test and build configuration code is already in wolfSSL.

K.1 Getting and Installing wolfSM

K.1.1 Get wolfSM from GitHub

Clone the wolfSM repository from GitHub:

```
git clone https://github.com/wolfssl/wolfsm.git
```

K.1.2 Get wolfSSL from GitHub

wolfSSL is needed to build and test the SM algorithm implementations. Checkout the wolfSSL repository from GitHub beside wolfSM:

```
# Directory structure should be:
# <install-dir>
# ├── wolfsm
# └── wolfssl

cd .. # To directory containing wolfsm
git clone https://github.com/wolfssl/wolfssl.git
```

K.1.3 Install SM code into wolfSSL

To install the SM code into wolfSSL, use the install script:

```
cd wolfsm
./install.sh
```

K.2 Building wolfSM

Once the wolfSM files have been installed into wolfSSL, you can build SM algorithms into wolfSSL.

Choose which algorithms you require on the configure line: `* --enable-sm3 * --enable-sm4-ecb * --enable-sm4-cbc * --enable-sm4-ctr * --enable-sm4-gcm * --enable-sm4-ccm * --enable-sm2`

For example, to include SM3, SM4-GCM and SM2:

```
./autogen.sh
./configure --enable-sm3 --enable-sm4-gcm --enable-sm2
make
sudo make install
```

K.2.1 Optimized SM2

To use optimized implementations of SM2 you can either use C only code or C code with the faster assembly code.

For C code only: `--enable-sp` For C and assembly code: `--enable-sp --enable-sp-asm`

Optimized C code is available for 32 and 64 bit CPUs.

Assembly code is available for the following platforms: * Intel x64 * Aarch64 * ARM 32-bit * ARM Thumb2 * ARM Thumb

K.3 Testing wolfSM

To test that the SM ciphers are working use the following command:

```
make test
```

To benchmark the algorithms enabled:

```
./wolfcrypt/benchmark/benchmark
```

To benchmark specific algorithms, add to the command line the option/s matching the algorithm/s: * SM2: `-sm2` * SM3: `-sm3` * SM4: `-sm4` or * SM4-CBC: `-sm4-cbc` * SM4-GCM: `-sm4-gcm` * SM4-CCM: `-sm4-ccm`

K.3.1 Testing TLS

SM ciphers are able to be used with TLSv1.2 and TLSv1.3.

Note: SM2, SM3 and at least one SM4 cipher must be built in order for SM ciphers suite to work. All algorithms must be SM.

The cipher suites added are: - ECDHE-ECDSA-SM4-CBC-SM3 (TLSv1.2, `--enable-sm2 --enable-sm3 --enable-sm4-cbc`) - ECDHE-ECDSA-SM4-GCM-SM3 (TLSv1.2, `--enable-sm2 --enable-sm3 --enable-sm4-gcm`) - ECDHE-ECDSA-SM4-CCM-SM3 (TLSv1.2, `--enable-sm2 --enable-sm3 --enable-sm4-ccm`) - TLS13-SM4-GCM-SM3 (TLSv1.3, `--enable-sm2 --enable-sm3 --enable-sm4-gcm`) - TLS13-SM4-CCM-SM3 (TLSv1.3, `--enable-sm2 --enable-sm3 --enable-sm4-ccm`)

K.3.1.1 Example of using SM cipher suites with TLSv1.2 An example of testing TLSv1.2 with "ECDHE-ECDSA-SM4-CBC-SM3" cipher suite:

```
./examples/server/server -v 3 -l ECDHE-ECDSA-SM4-CBC-SM3 \
    -c ./certs/sm2/server-sm2.pem -k ./certs/sm2/server-sm2-priv.pem \
    -A ./certs/sm2/client-sm2.pem -V &
./examples/client/client -v 3 -l ECDHE-ECDSA-SM4-CBC-SM3 \
    -c ./certs/sm2/client-sm2.pem -k ./certs/sm2/client-sm2-priv.pem \
    -A ./certs/sm2/root-sm2.pem -C
```

The output using the commands above will be:

```
SSL version is TLSv1.2
SSL cipher suite is TLS_ECDHE_ECDSA_WITH_SM4_CBC_SM3
SSL curve name is SM2P256V1
SSL version is TLSv1.2
SSL cipher suite is TLS_ECDHE_ECDSA_WITH_SM4_CBC_SM3
SSL curve name is SM2P256V1
Client message: hello wolfssl!
I hear you fa shizzle!
```

K.3.1.2 Example of using SM cipher suites with TLSv1.3 An example of testing TLSv1.3 with "TLS13-SM4-GCM-SM3" cipher suite:

```
./examples/server/server -v 4 -l TLS13-SM4-GCM-SM3 \
    -c ./certs/sm2/server-sm2.pem -k ./certs/sm2/server-sm2-priv.pem \
    -A ./certs/sm2/client-sm2.pem -V &
./examples/client/client -v 4 -l TLS13-SM4-GCM-SM3 \
    -c ./certs/sm2/client-sm2.pem -k ./certs/sm2/client-sm2-priv.pem \
    -A ./certs/sm2/root-sm2.pem -C
```

The output using the commands above will be:

```
SSL version is TLSv1.3
SSL cipher suite is TLS_SM4_GCM_SM3
SSL curve name is SM2P256V1
SSL version is TLSv1.3
SSL cipher suite is TLS_SM4_GCM_SM3
SSL curve name is SM2P256V1
Client message: hello wolfssl!
I hear you fa shizzle!
```

COPYRIGHT ©2024 wolfSSL Inc.